UNIVERSITY
OF OSLO

Master's thesis

Reinforcement learning with the TIAGo research robot

Manipulator arm control with actor-critic reinforcement learning

Markus Toverud Ruud

Robotics and Intelligent systems
60 ECTS study points

Department of Informatics
Faculty of Mathematics and Natural Sciences

Spring 2023

Markus Toverud Ruud

Reinforcement learning with the TIAGo research robot

Manipulator arm control with actor-critic reinforcement learning

Abstract

Control of robotics for object grasping and manipulation is still a complex problem with many different approaches and solutions. This project examines the usage of actor-critic reinforcement learning methods in an attempt to teach reinforcement learning agents to control a robotic manipulator arm attached to a mobile research robot. The agent controlling the arm is trained to attempt to reach for and assume a pre-grasp position around an object placed on a table in a simulated world. Different agents are trained with various degrees of utilization of the robot's sensory systems and with varying definitions of the parameters associated with reinforcement learning, such as the state space, action space, and reward function definitions. Comparative experiments are conducted in the same simulated environment, comparing the different reinforcement learning agents with each other, as well as comparing the best-performing agent with a traditional motion planning algorithm. The results indicate that the best-performing agent's proficiency at reaching and assuming a pre-grasp position in its initial simulation environment (PyBullet) peaks at 92%. However, when transferred to and retrained in the same environment in which the motion planning algorithm is implemented (Gazebo), its accuracy reduces significantly to 55.8%. In contrast, the motion planning algorithm achieves a success rate of 76% for the same task. Although the results suggest that the best-performing trained agent's accuracy is worse than that of the traditional motion planning algorithm for the task presented in the Gazebo environment, it is significantly faster at reaching the object because it does not require pre-planning the motion. Additionally, different simulation environments are discussed, highlighting the differences between using Gazebo, the most common simulation environment for robotics development, and PyBullet a physics engine implemented for ease of use in Python.

Contents

List of Figures

List of Tables

Preface

This master's thesis is submitted to the Department of Informatics at the University of Oslo as part of the author's master's degree in the study program Informatics: Robotics and Intelligent Systems. The main supervisor of this project is Professor Jim Tørresen, group leader of the Robotics and Intelligent Systems (ROBIN) research group at the Department of Informatics at the University of Oslo. The work is partially supported by The Research Council of Norway (RCN) as a part of the projects: Vulnerability in the Robot Society (VIROS) under grant agreement no. 288285 and Predictive and Intuitive Robot Companion (PIRC) under grant agreement no. 312333.

Aknowledgements

I would like to sincerely thank my parents for letting me live in their basement while writing this thesis and not having to endure the economic strain of being a student in the Oslo rent market.
I would also like to thank my co-students and the staff at the ROBIN research group for providing a good study environment.

Chapter 1

Introduction

The fields of robotics and AI (artificial intelligence) powered systems have seen formidable growth in recent years. Researchers all around the world are attempting to figure out how to best combine the fields to utilize the promising results of AI-based methods to create and control robots that can be of great utility in a variety of tasks.
In Norway, there is a growing lack of workforce personnel in vital roles in society, such as health care professionals. According to estimates by the Norwegian Department of Health in 2021, different fields of healthcare are understaffed by approximately 10,000 workers [19]. Part of a solution to issues like this is theorized to be found in automation, using robots and other autonomous or semi-autonomous machinery to assist with and reduce the workload and number of personnel needed to provide adequate care to patients. Development is generally split into two main categories, affective and effective [42], concerning either caring for the emotional well-being of the user or assisting with repetitive or strenuous physical tasks. At this point in time, no cost-effective solution exists to the problem, yet a variety of robotic research platforms are being developed to find and research solutions to it. One such research platform is the TIAGo robot, recently purchased by the robotics and intelligent systems research group (ROBIN) at the University of Oslo. Falling mostly into the effective category of robot development platforms, TIAGo features a mobile base, a manipulator arm, a variety of sensory systems, and some pre-programmed functionality, provided by the manufacturers of the robot. The platform serves as a great foundation to research AI-based manipulator control methods. Of those methods, reinforcement learning (RL) is a promising approach to address complex manipulator control tasks, as it allows the robot to learn from its own experiences and improve its performance over time. Developments in RL research have produced methods that can be utilized for control problems in dynamic, continuous environments, and can thus possibly be harnessed to approach complex real-world scenarios.
This project is based on attempting to utilize the TIAGo research platform.

With object manipulation in mind, an attempt is made to wield and train reinforcement learning agents to control TIAGo's arm in relation to objects on a table. It is planned with the width of ROBIN's study program in mind, to utilize both machine learning techniques and some robotics expertise in the same project.

1.1 Research goals

Throughout this project, two core questions are being studied.

Q1: Is it possible to utilize actor-critic reinforcement learning techniques to facilitate object picking with the TIAGo robot?

Q2: How does the resulting behavior of the reinforcement learning techniques compare to a motion planning approach already implemented for the robot?

Even though techniques and algorithms already exist for object handling and manipulation in TIAGo's packages, none of them include controllers that are completely machine-learning-based. The approach in this project is utilizing reinforcement learning techniques combined with varying degrees of sensory feedback to teach the agent controlling the arm to regulate the arm's movements on the go while the arm is in motion. While working on a problem that is already solved by classic trajectory and motion planning techniques, it is worth addressing through a reinforcement learning perspective as it may prove to solve the problem in novel or interesting ways.

1.2 Scope and limitations

In this thesis some somewhat unsuccessful attempts are first made to teach the agent controlling the arm to learn to dynamically reach for objects on a table, utilizing depth information gathered by TIAGo's depth camera to simultaneously avoid collision with the table itself and other objects it is not supposed to reach for. All the initial attempts are conducted in the Gazebo simulation environment [26], a simulation environment that is commonly used for robotics simulations and development because of its close ties to ROS (Robot Operating System) [36], the most common platform for development of robotics. Later, the simulation environment is replaced by the PyBullet [9] environment, in which agents are trained more successfully, but lack any sense of collision detection, as simulation of the depth camera of the robot is not implemented. The agents in the new environment learn to control the arm to reach for objects and are trained to become successful at reaching them, and semi-successful at reaching and assuming a pre-grasp position. By successfully reaching a pre-grasp

position the object is not displaced by the arm's movements and the end-effector grippers of the arm end up around the object, ready to grasp it. The trained agents of the PyBullet environment are also evaluated in the Gazebo environment, to observe how the trained skills transfer directly from one simulated environment to another. The best-performing agent is also retrained in the Gazebo environment and compared to an example motion planning algorithm, provided in the software packages of the robot. The topics of object detection and position estimation are also briefly addressed in the context of the reinforcement learning agents presented.

1.3 Outline

First, some background information will be presented, starting with some general information about robotics and robotic development. Moving on to the fundamentals of robotic modeling and control, with topics such as coordinate frame assignment and set-point control. In addition, trajectory and motion planning will be introduced as a comparison to AI-based control. Moving on to the basics of machine learning and going a little more in-depth on the topic of reinforcement learning.
After the background section, the methods used in the practical applications will be introduced, including the most useful ROS libraries, the setup for the reinforcement learning agents, and the reasoning for the choices of state spaces, action spaces, and reward functions designed for the tasks presented. The specific reinforcement learning algorithms used by the agents are also presented on a surface level. Lastly, a naive object position estimation procedure is presented, utilizing an object detection package that has been created for use in ROS.
In the preliminary experiments section, the first rounds of simulated experiments are presented, including the somewhat unsuccessful attempts and the thought process moving forward from them.
In the results section, the main results are presented, including the agents trained in the PyBullet environment, their performances when directly transferred to the Gazebo environment, the performance of the retrained agent, the comparison between the retrained agent and an example motion planning algorithm, and lastly a quick evaluation of a naive object position estimation procedure.
In the discussion section, the results are discussed along with discussions about the choice of simulation environments, safety concerns of the trained agents, proposed future work, and ethical considerations.
In the appendix, some technical specifications concerning the software packages utilized in this project are presented, along with some quick tutorials on how to run the trained agents in the Gazebo environment.
Appendices on select topics required to understand robot modeling and

TIAGo's specific constraints are also included, along with example motion sequences generated by the best-performing reinforcement learning agent and the example motion planning algorithm.

Chapter 2

Background

2.1 Robotics in society

2.1.1 The industry of modern robotics

Making robots and other intelligent systems do simple day-to-day tasks in a safe and sustainable manner is still an unsolved problem for many tasks and use cases. The market for privately owned robotics is growing, yet the trend seems to go towards more specialized devices such as robotic lawnmowers, vacuum cleaners, and window cleaning systems and not fully- or semi-autonomous service robots [43].
A care robot is by definition a machine that can do tasks related to physical or emotional care, either autonomously or at least semi-autonomously [14]. The vision of having care robots to facilitate elderly caregiving and patient rehabilitation is not a new one. Scientific research teams from all over the world have been working on the problem for years. Some working development platforms have been made such as the 'Care-O-bot'[2], Hector [20], and the HOBBIT project [11], yet none of these prototypes has led to any commercially available product, despite some of the projects' over two-decade longevity. There are still roadblocks in terms of having functional robotic units in health care, such as public spending, as some of the most modern development solutions are quite expensive and of course, most important of all, safety standards. The need to meet safety regulations has led designers to simplify their goals, as simplified goals are easier to accomplish and monitor. It has also split development into two main categories, 'effective' and 'affective', in which the effective branch focuses on the utility of the care robot, such as fetching items, helping with loads, and cooking, while the affective branch focuses on emotional support, such as robotic pets or interactive screens that a user can communicate with.

2.1.2 Human-robot interaction

An important topic when considering the implementation and deployment of robotics is human-robot interaction. HRI is in broad terms how the machine interacts with and is perceived by individuals in its surrounding environment. The field can range from topics such as safety of operation, as well as robotics' broader impact on society, as jobs have been and will continue to be replaced by computing machinery. HRI also involves the acceptance of relying on computer systems in settings where a person would previously be in full control, such as with fully autonomous vehicles. It can also involve rules, regulations, and norms created by society to regulate and guide to which extent robotic units are allowed to operate.
In an article from 2016 [38], some of these topics are discussed. It is evident that on some fronts, technology already exists to replace certain jobs. For instance, computer systems that negate the need for a copilot on a commercial aircraft or the coming of fully autonomous vehicles. However, public opinion on the safety of these systems is somewhat skewed from that of professionals and researchers in the fields as some of the technologies are viewed as somewhat 'risky' and more dangerous than they truly are. It is also questioned if it might be more advantageous for some tasks to develop systems that greatly aid and make jobs easier than replacing them entirely. Systems that augment the user such as radar-augmented cruise control, run-off-the-road alarms, and vehicle-to-vehicle communication in cars does a good job of keeping traffic more safe, while still keeping the human driver.
In the context of health care and rehabilitation, many back-breaking tasks have to be performed, such as carrying and moving around injured or elderly patients. It is found that workers in the field spend much of their time doing manual labor or administrative tasks instead of taking care of the emotional needs of the residents/patients[27]. It would thus be very helpful to have some assisting systems in place so that they could spend their time making sure the residents/patients have their emotional needs met rather than spend most of their time taking care of the most basic physical needs.
A proposed solution to this is having a robotic assistant take care of the most repetitive and basic tasks, such as cleaning, cooking, carrying items, and patient checkups. However, tasks like that require very complex manipulation skills, and it requires the skills to be utilized in very dynamic and differing environments. Effective motion planning algorithms already exist, but often rely on lengthy planning steps before executing any action, and are often dependent on the environment being static while moving or the planned action will fail. Instead, AI agents could be used to control the movements of the robotic manipulators, possibly allowing them to learn how to solve complex tasks in dynamic environments.

2.1.3 The TIAGo research robot

TIAGo is a mobile service robot designed to work in indoor environments. It features an extendable torso, a manipulator arm designed to grab objects, and a mobile base. The arm can be fitted with several different gripper options and has seven degrees of freedom(7-DOF), and can handle a payload of up to 3 kg. It features a sensor suite that allows it to perceive, plan and navigate in its own environment through the use of lasers, sonars, and a depth camera. It also comes with ROS(Robot Operating System) integration.
Because of its purpose as a research platform, it is also highly customizable and can be controlled in a multitude of ways. A plethora of tutorials exist for free on high-level control of the robot and its Gazebo simulation implementation is freely available to anyone interested in testing and developing software for the platform. While the software packages for TIAGo include navigation packages for self-localization and mapping (SLAM), obstacle avoidance, and topological localization, they are not utilized in this project. Instead, lower-level control such as set-point control for the manipulator arm is used and integrated into AI-based control, as it allows the reinforcement learning agent to have total control of the manipulator arm.

2.2 Robot perception and scene understanding

Robot perception systems are generally a combination of input sensory data that feeds into an AI/Machine Learning (ML) model that utilizes the sensory data stream for decision-making or control. The integration of sensory data is a crucial component of multiple functions such as object detection, scene understanding, and more [33]. It is also important to recognize that different kinds of robotic systems utilize a variety of different sensors, manufactured by different companies, resulting in differences in sensory readings and utilization. In the case of environmental mapping, an indoor robot can usually naively assume flat terrain, and thus does not have to include intricate systems for calculating the terrain it has to move upon. For outdoor robots it is quite the opposite case, it has to include very robust systems to map its sensory data to a complex terrain if it is to succeed in its tasks. Environmental conditions also have to be considered when implementing a robotic system. A sensor might work very well in indoor lighting or in broad daylight but struggles when reading values if lighting conditions are poor. It is thus important to know which sensors to use in certain situations and why. In the context of object grasping and manipulation, the visual sensory systems are most important as they are vital in the detection, location, and tracking of objects.

2.2.1 TIAGo's visual sensory systems

The TIAGo research robot is equipped with an RGB-D camera. The camera works by processing two separate image streams. The first image stream captures a standard RGB image, while the second captures a depth image or point cloud, providing depth information throughout the camera's field of view. The combination of the two streams enables advanced computer vision techniques, such as self-localization and mapping [10]. In addition, they also allow for the deployment of state-of-the-art object detection models on the RGB image, which can be projected on the depth image to determine the position of objects in the world, allowing for precise object manipulation.

2.3 Robotic modeling and control

In order to fully comprehend the decision-making process of designing the reinforcement learning agents trained in this project and how they control the robot, it is necessary to possess a basic understanding of how modern robotics are modeled and implemented, as well as an understanding of how each joint is controlled. To compare with how the reinforcement learning agents operate, it is also important to know the basics of how motion planning algorithms are utilized.

2.3.1 Rigid body transformations/Coordinate frame assignments

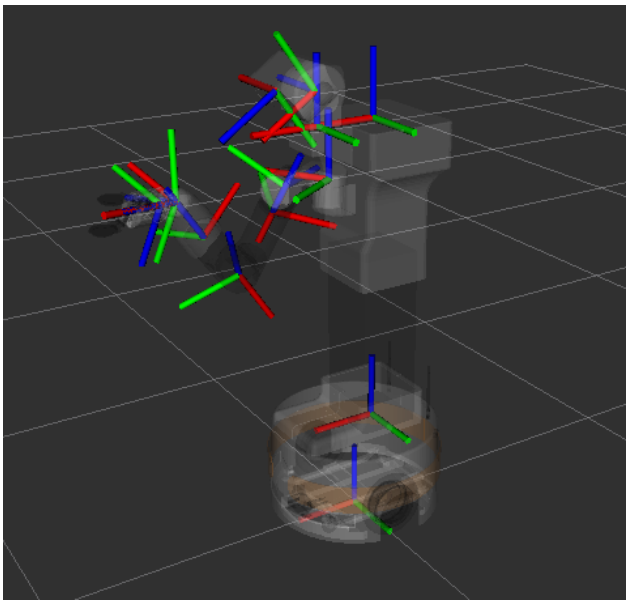


Figure 2.1: Select coordinate frames of the TIAGo robot

Modern robots are usually modeled as a chain of reference coordinate frames that are designed in such a way that a transformation from one frame to the next in the chain only involves one dynamic variable. This can be achieved by following the Denavit-Hartenberg conventions [40]. These conventions state that the homogenous transformations between one coordinate frame and the next can be characterized by first, a rotation around the z-axis, then a translation along the z-axis, a translation along the x-axis, and lastly a rotation around the x-axis. Transforming in this order results in a matrix A that describes the homogenous transformation in only four variables. When used in the context of robotics only one of these variables is usually dynamic and controllable, and the other three are static and determined by the physical design of the robot.
By having many 'simple' matrices describing the transformations from one frame to the next, a tree of interlinked coordinate frames can be built, describing the chain of transformations required to transform from one node of the tree to another. Some of TIAGo's assigned coordinate frames can be viewed in fig. 2.1 and most of TIAGo's coordinate frame tree can be viewed later, in fig. 3.2. Having such a tree removes the necessity to transform through each and every coordinate frame by sequentially multiplying the homogenous transformation matrices with each other. If, for instance, a depth camera is utilized that captures a point P_{cam} in the reference frame of the camera and it is valuable to know the point in relation to the base of the robot, and not the camera, a transformation matrix can be built using the tree. Assume there are three reference frames, one for the camera, an intermediate reference frame in the middle, and one for the base of the robot. The transformation from the reference frame of the camera of the robot to the intermediate frame can be characterized by the homogenous transformation $A_{cam}^{intermediate}$ and the transformation from the intermediate frame to the reference frame of the base of the robot can be characterized by the homogenous transformation $A_{intermediate}^{base}$. The direct homogenous transformation from the reference frame of the camera to the reference frame of the base of the robot can then be found as the transformation matrix $T_{cam}^{base} = A_{intermediate}^{base} A_{cam}^{intermediate}$. The point P_{cam} can then easily be transformed to be described in the base reference frame by first padding the point and then multiplying the transformation matrix T_{cam}^{base} with it.
In practice, this is called forward kinematics, when joint angles or displacements are known and used in homogenous transformations to calculate the positions of the robotic links in three-dimensional space and in a common reference frame. The joint angles and linear displacements are readily available through proprioceptive sensors [39] such as ring encoders, which read and keep track of the robot's prismatic and revolute joints. See appendix B for a practical example.

2.3.2 Set-point control

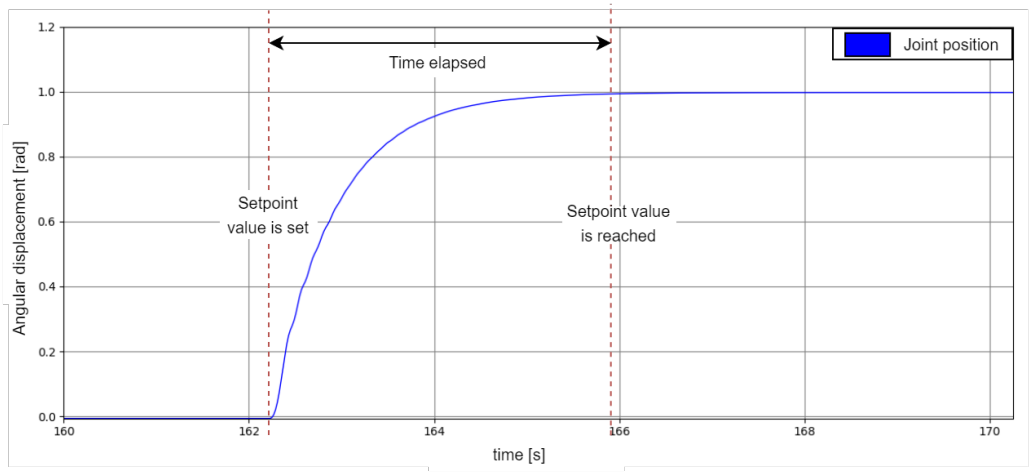


Figure 2.2: Example set-point controller step-response

In the context of manipulator control, it is possible to control each joint by setting a desired position, velocity, acceleration, or torque/force. However, only one parameter can be regulated at a time. To do so requires the use of a set-point controller. A set-point controller is a controller that regulates the system's output to a desired or set-point value. It does so by continually measuring the system's output and comparing it to the set-point value, generating an error.

$$Error = desired\ value - measured\ value$$

The controller then adjusts the system's effort in order to minimize the error and bring the output closer to the input. The controller does so by utilizing control systems such as proportional-integral-derivative (PID) control to regulate the response of the system to the input signal. A controller such as this usually has to be manually tuned to achieve critical damping, where the system reaches the desired value as quickly as possible without overshooting and oscillations. With TIAGo, the set-point controller parameters are already pre-configured and can directly be utilized by motion planning algorithms or AI agents. In fig. 2.2, the response curve of setting a desired angle for one of TIAGo's revolute joints can be observed.

2.3.3 Trajectory and motion planning

The goal of trajectory planning is to produce a sequence of reference inputs to the manipulator such that it executes a planned motion while also considering the timing of the movements. The act of planning a motion requires several stages. First, a path has to be determined, which represents

the desired trajectory of the end-effector of the manipulator. The path can be designed as a continuous polynomial or a sequence of connected points in space. The operating space of the manipulator is a crucial factor when the path/trajectory is to be determined. The operating space describes the permissible or reachable positions in the space the manipulator occupies, and thus constraints and limits the motion the manipulator physically can perform. The planned motion has to be within the manipulator's operating space for the entirety of the planned motion.
The next part of the planning consists of inverse kinematics. Inverse kinematics is the act of figuring out the necessary joint space configurations to reach a point in space with the end-effector. In most cases, a single point in space in the manipulators' operating space can be reached with multiple different configuration space parameters. This assumption holds for most permissible configurations in a manipulator's operating space unless the point is at the border of the space. In these scenarios the manipulator has to be fully extended to reach the point, resulting in what is called a singularity, a configuration where the robotic manipulator loses one or more degrees of freedom.
After establishing a permissible path and identifying the necessary configuration space parameters to reach the desired end position through inverse kinematics, functions for each controllable joint can be described. For an arm of n controllable joints, each function $q_n(t)$ describes the set-point value required of joint n at time t to reach the final, planned configuration. Each function $q_n(t)$ can thus be used as the input of the set-point controller controlling joint n, causing the joints to actuate.

2.3.4 Motion planning algorithms

In the previous section, the process of generating a trajectory is discussed. However, it is often a challenging task to find a suitable path from the starting configuration of a manipulator to the goal configuration, requiring complex algorithms and a detailed model of the environment. A general way to model the environment is to start with the manipulator's configuration space C, containing all permissible configurations $\mathbf{q} \in \mathbb{R}^n$, where n is dictated by the dimension of the configuration space which usually coincides with the number of controllable parameters (prismatic or revolute) of the manipulator. Obstacles are added to the configuration space. A real-world obstacle's image in the configuration space is called CO_i and is a subspace of C in which all configurations $\mathbf{q}$ cause collision between the manipulator and an obstacle O_i. CO_i regions are also defined as regions in which the manipulator collides with itself, making them inaccessible. The subspace CO of C is the union of all obstacles images and regions of self-collision in the manipulator's configuration space.

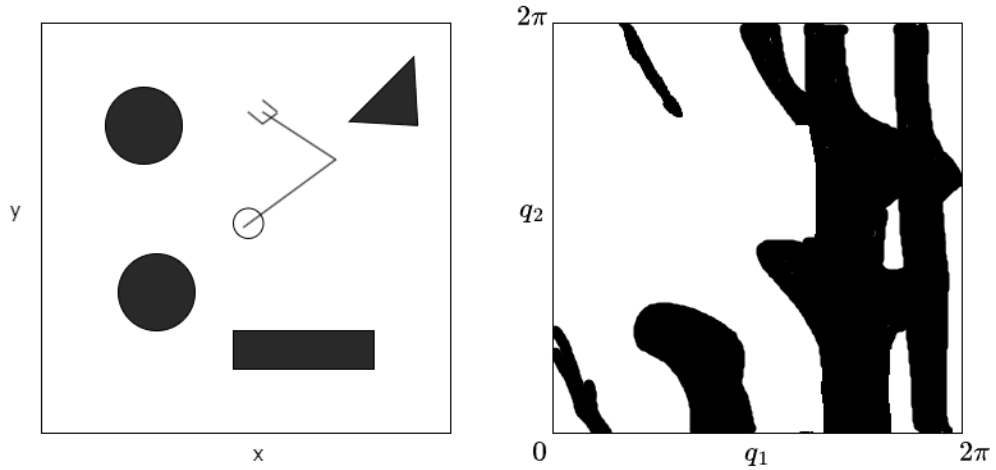


Figure 2.3: **Left:** 2-link manipulator in an environment with obstacles in a planar world. **Right:** The configuration space of the same world where the blacked-out regions are *CO* obstacle regions. *Adapted from [39].*

With the definition of the configuration space C and the obstacle region CO, a subspace $C_{free} = C - CO$ is defined, a region that is free of obstacles and in which configurations $\mathbf{q}$ has no collisions. A goal for the motion planning algorithm can thus be defined. For a starting configuration $\mathbf{q}_s$ and a goal configuration $\mathbf{q}_g$, find a continuous path through C_{free} from $\mathbf{q}_s$ to $\mathbf{q}_g$.

Sampling based methods

Probabilistic motion planning methods are algorithms that are useful for efficiently solving high-dimensional configuration space problems. As a manipulator's configuration space increases in dimensionality and thus complexity, it becomes too computationally expensive to exhaustively search the whole space for a roadmap from the starting configuration to the goal configuration. Sampling-based methods avoid having to search the whole space by randomly sampling configurations q from C and checking for collisions to build up an adequate image of C_{free}. The sampled configurations can be used in different ways, depending on the algorithm. Probabilistic Roadmap (PRM) uses the sampled configurations to build a graph structure approximation of the whole C_{free}, connecting each new collision-free configuration with its k closest neighbors and when a large enough area of C is searched, uses a search algorithm such as Dijkstra's algorithm to search the graph and find the shortest path from the starting configuration to the goal. Other sampling-based

methods, such as Rapidly-exploring random tree (RRT) use the sampled collision-free configurations to grow a space-filling tree, terminating when a path is found from the starting configuration to the goal configuration. While both methods are probabilistically complete, guaranteeing that if a path exists from the starting to the goal configuration, it will be found, they vary in efficiency. PRM is a multi-query method, meaning that the approximation of C_{free} generated by using the method can be reused to find other paths as long as the obstacles in the configuration space remain static. RRT, on the other hand, is single-query, requiring a new search of C each time a new path is to be found, it is however, much faster than PRM when called for only a single path. Additionally, it is worth noting that optimal versions of both PRM and RRT have been developed [24], which guarantees finding the optimal path.

2.4 Machine leaning

To understand the learning process of reinforcement learning agents, it is important to know the fundamentals of neural networks and their iterative updating process.

2.4.1 Neural networks

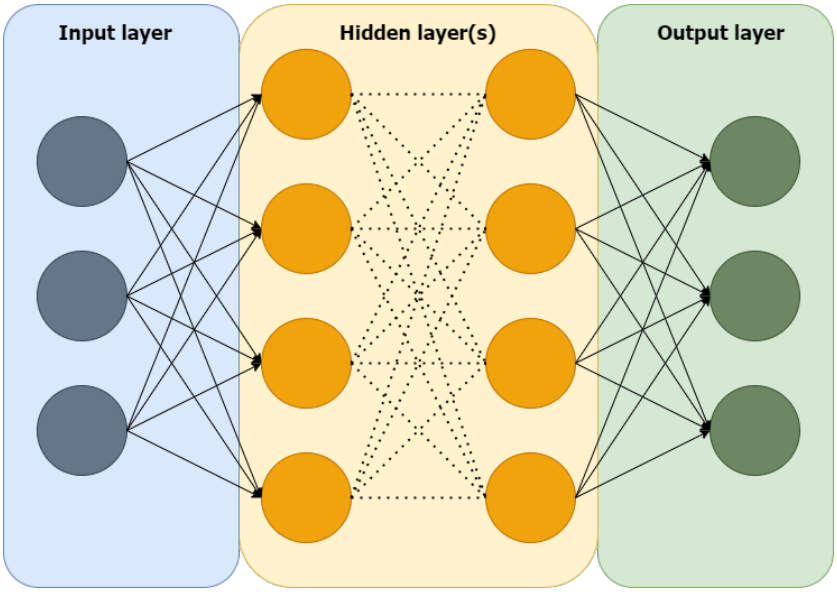


Figure 2.4: Simple neural network illustration

Neural networks are computational models, used for decision-making or predictions based on information they receive. A neural network must always consist of an input and an output layer but requires one or more

hidden layers to be considered 'deep'. Deep neural networks are primarily split into three main parts, an input layer, a hidden layer or multiple hidden layers, and an output layer. The input layer receives some sort of information, known as features, similar to covariates in a statistical model. The features are numerical values that represent information about what the network is attempting to predict, or often in the case of reinforcement learning, information about the environment an agent operates in.
The hidden layer or layers contain nodes that sum up all the input nodes and applies a function to the result, then pass the result into the next layer. It is also common to see some form of bias b applied to the summation. The output of the summation in each node is as follows:

$$\hat{y} = \sum_{i=1}^{n} w_i x_i + b$$

Where n is the number of inputs going into the node, x_i is the input value for input i, w_i its trainable, scalar weight, and b is the optional bias term. After the summation, a function is applied to the sum, called the activation. The function most commonly used as an activation function for a hidden layer is the ReLU (Rectifying linear unit) function or a leaky ReLU. The output of a neuron in the hidden layer with a ReLU activation function is given as:

$$y = f(\hat{y}) = max(0, \hat{y})$$

The output layer receives the outputs of the last hidden layer and produces a prediction/decision based on the information processed through the network. Each node in the output layer functions similarly to the nodes of the hidden layers except that the activation function used is selected according to the specific usage of the network. For instance, if the network is to be interpreted as a classifier, the output could be the percent-confidence of the network that the presented data belongs to a certain class and thus the output layer needs to use an activation function that squishes the output to be between 0 and 1, this can be achieved by using a sigmoid activation function for binary classification or softmax for multi-class.

2.4.2 Neural network weight updates

The purpose of neural networks is to learn patterns and relationships in the data so that they can be used as predictors, either for classification or as shown later, for a reinforcement learning agent to choose actions for a given state in an environment. To achieve this, the network needs a method to update the weights between the nodes of its layers (denoted w_1, w_2, w_3 in fig. 2.5). The update process is most clearly defined in what is called supervised learning. In supervised learning, the network is trained with labeled data, in which the correct output of the network is known. Because the correct output is known, the actual output received

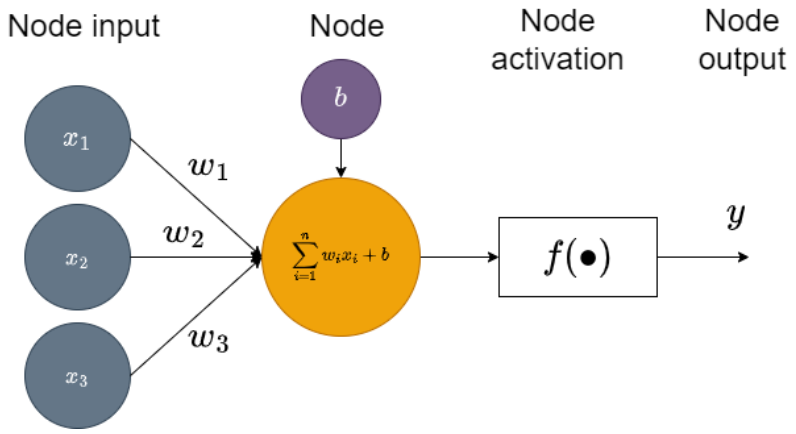


Figure 2.5: Simple neural network neuron illustration

from the network is directly comparable to it. Because of this fact, a loss function can be used to calculate the accuracy of the network's output. The choice of loss function depends on what kind of network is being trained and what kind of prediction or classification the network is doing. Some common and useful loss functions include MSE(mean square error), which is the summed square difference between the true values of the output and the values generated by the network, CE-loss(cross-entropy loss), which calculates the difference in each class or labels probability contrary to the true labels or classes, and KL-divergence(Kullback–Leibler divergence), which is a measurement of how two probability distributions are different from one another. For a network of N output layer nodes with outputs $\hat{y}_i,\ i \in \{1, ..., N\}$ and corresponding ground truths $y_i,\ i \in \{1, ..., N\}$, some of the most common loss functions are as follows:

$$MSE = \sum_{i=1}^{N} (\hat{y}_i - y_i)^2,\ CE - loss = -\sum_{c=1}^{N} y_i \log(\hat{y}_i),\ KL - div = \sum_{c=1}^{N} \hat{y}_c \log \frac{\hat{y}_c}{y_c}$$

The two main purposes of a loss function are:

1. *To measure to which degree the prediction of the neural network is correct.*
2. *To be used to update the weights of the networks in a manner that enables future predictions to be more accurate.*

In order for a network to improve its predictions, a backpropagation algorithm is utilized. This algorithm starts at the output of the network and calculates, through the gradient of the loss function, how much it should adjust the weight of each node connected to it. This process is then repeated for every layer until the network input. For a weight w of the network, the prediction $\hat{y}$, and the error E of the prediction, calculated

through the loss function, find:

$$\frac{\delta E}{\delta w} = \frac{\delta E}{\delta \hat{y}} \frac{\delta \hat{y}}{\delta w}$$

And update the weight by:

$$w = w - \eta \frac{\delta E}{\delta w}$$

The update is scaled by a learning rate η, which determines how much the weight is allowed to update at a time and often decreases during training. The decrease is to ensure that the update process becomes deterministic and the weights converge over the duration of the training. The learning rate is often high at the beginning of training, resulting in large changes to the weights of the network, which causes large and quick performance increases, but makes it unable to fine-tune to optimal weight values. When the rate is later adjusted down, the performance increases also slow down while the network parameters are being fine-tuned. Most modern neural networks also use optimization algorithms, which often store a decaying average of the past partial derivatives of every weight w to smooth out the large parameter jumps caused by using only the current gradient of the loss and to give momentum when the current partial derivatives become very small.

2.5 Reinforcement learning

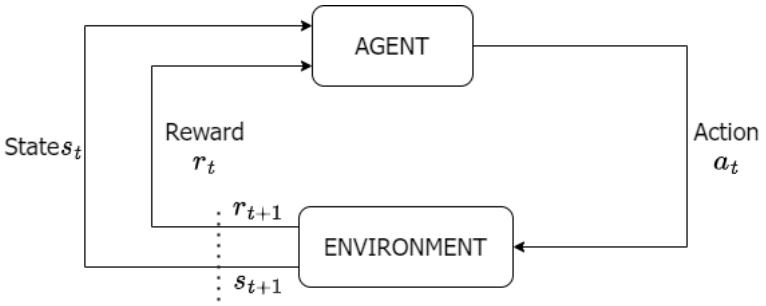


Figure 2.6: Reinforcement learning illustration

Reinforcement learning(RL) [41] is a machine learning approach used to teach systems how to learn a policy by exploring the system's environment and optimizing its behavior over time, through trial and error. The policy is a function that maps states of the environment to actions taken by the agent, thus dictating an agent's behavior. The system receives feedback from the environment in the form of rewards, which can either be negative, a punishment for performing an action that is

deemed as bad for the current state of the agent in the environment, or positive, a reward for performing an action that is deemed good for the current state of the agent in the environment. The reinforcement learning agent uses the feedback received in the form of rewards to mathematically optimize its choice of actions over time and thus learns how to better act in situations similar to or identical to situations it has encountered before. The environment itself is described in the form of a Markov decision process(MDP). A mathematical framework used for decision-making problems. Combining reinforcement learning with deep neural networks has not only made it possible for reinforcement learning algorithms to learn from large and complex data such as pixel values from video games or live camera feeds [32], but also learn to optimize behavior in continuous environments, in which the environment cannot be described by a finite number of states. Realistically, many complex real-world problems, such as manipulator control, cannot easily be abstracted and thus require the environment to be continuous. Some of the most popular and well-known reinforcement learning algorithms are Q-learning [44] and SARSA [35].

2.5.1 The Markov decision process

A Markov decision process is a discrete-time stochastic control process [34], that is useful for studying optimization problems like reinforcement learning. It is mathematically stated as tuple $M = (S, A, P_a, R_a)$, in which S is a set of states $s \in S$ called the 'state space' or 'observation space' in reinforcement learning, A is a set of actions $a \in A$, called the 'action space' in reinforcement learning, $P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a in state s at a time t leads to a state s' at time $t + 1$ and $R_a(s, s')$ is the reward or expected reward from going from state s to state s' due to action a. The goal of a Markov decision process is to find an optimal policy π, a function $\pi(s) = a$ that finds the optimal action a at a given state s. The discrete-time nature of Markov decision processes can make them harder to apply to real-world control problems because the problems may require continuous control signals. If a continuous control signal is to be determined via a Markov decision process such that the control signal $q(t) = a(t)$, it can be troublesome because of the Markov decision process's inability to produce a continuous function $a(t)$. However, the issue can be somewhat mitigated by sampling the environmental state s_t and receiving the optimal action a_t from the decision process at a high rate, causing the discrete set of actions $\{a_1, ..., a_n\}$ to approximate a continuous action signal $a(t)$.

2.5.2 Model-based and model-free free reinforcement learning

The term 'model', in reinforcement learning, refers to whether or not the agent utilizes or learns an explicit model or distribution of the environment, that has all the transition dynamics and probabilities calculated [21]. In these cases, the agent can quickly sample this distribution and pick the state transition that gives the most expected future reward, e.g. picking a move in a chess game that can be calculated to have the highest probability of leading to a win later in the game. For instances where no such model of the environment exists, the reinforcement learning agent can attempt to build or approximate its own model of the transition dynamics and probabilities of the environment, labeled as 'model-based' reinforcement learning algorithms. However, methods of this family require the agent to adequately learn how to predict how every available action a_t in every possible state s_t leads to the next state s_{t+1}. A task that is doable for environments that are well-understood and predictable, such as board games, in which it is trivial to predict the state of the board after performing an action. On the other hand, actions in real-world environments, in which both the action and state spaces are continuous, are much less predictable and thus it would require immense computational power and incredibly large models to accurately predict the next state s_{t+1} for all available actions a at a given state s_t. Instead, 'model-free' approaches can be utilized to negate the need for future predictions.
Model-free reinforcement learning methods are typically categorized into two slightly different approaches:
1. Policy optimization: The goal of policy optimization methods is to learn a policy function $\pi_\theta(s)$, with parameters θ, which outputs the best possible action a for a given state s. Updates to the function are typically performed 'on-policy', meaning that the parameters θ are only updated using data collected while acting according to the current policy.
2. Q-learning: The goal of Q-learning methods is to learn an action-value function $Q_\theta(s, a)$, with parameters θ, or table $Q(s, a)$, which outputs the expected return of choosing an action a at state s. Updates to the function parameters or table are typically performed 'off-policy', meaning that the updates to the parameters θ can be done using data collected at any point during training. The next action of the agent is then chosen by considering all available actions a at state s and finding:

$$a(s) = arg\max_a Q_\theta(s, a)$$

While both methods attempt to learn how to best pick an action a at a given state s, policy optimization methods do so by learning a policy directly, making them more stable but less efficient because the methods

cannot reuse old samples. Q-learning methods do not directly learn how to best choose actions for given situations, and the performance of the agent is only indirectly increased by learning the Q-function. However, since all training samples can be utilized, the methods are more efficient for learning but tend to be less stable and dependent on good exploration strategies.

2.5.3 Action-value functions and the bellman equation

In the previous section, an action-value function $Q(s, a)$ is presented. The action-value function adheres to what is called the Bellman equation[5]. In principle, the equation describes how the value of a state-action pair relates to the values of the next states and actions that can be taken. There are two slightly different definitions of the action-value function that adhere to this principle:
1. The 'on-policy' action-value function $Q^{\pi}(s, a)$, describing the value of taking an arbitrary action a in a state s and forever continuing to perform actions defined by the policy π for every next state the agent lands in.

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{s' \sim P}[r(s, a) + \gamma \mathop{\mathbb{E}}_{a' \sim \pi}[Q^{\pi}(s', a')]]$$

Where $s' \sim P$ indicates that the next state s' is sampled from the environment and $a' \sim$ indicates that the next action a' is determined by the current policy π. The term γ is called the discount factor and determines how much the expected future rewards should be weighted in relation to the immediate reward $r(s, a)$ received from the environment for performing the action a in state s.
2. The optimal action-value function $Q^{*}(s, a)$, describing the value of taking an arbitrary action a in a state s and forever continuing to perform the action that is deemed as most optimal in the given next states s_{t+n}.

$$Q^{*}(s, a) = \mathop{\mathbb{E}}_{s' \sim P}[r(s, a) + \gamma \max_{a'} Q^{*}(s', a')]$$

Where all the terms indicate the same as in the 'on-policy' definition, except the next action a' is determined as the action that maximizes future return, rather than adhering to a defined policy π.

2.5.4 Deep reinforcement learning

Deep reinforcement learning is the practice of combining reinforcement learning agents with deep neural networks. The deep neural network can serve multiple purposes. Approaches of this category are particularly useful when one would want the agent to learn directly from a complex data source such as a direct image feed. The process works because of the deep neural network's property of being able to find compact,

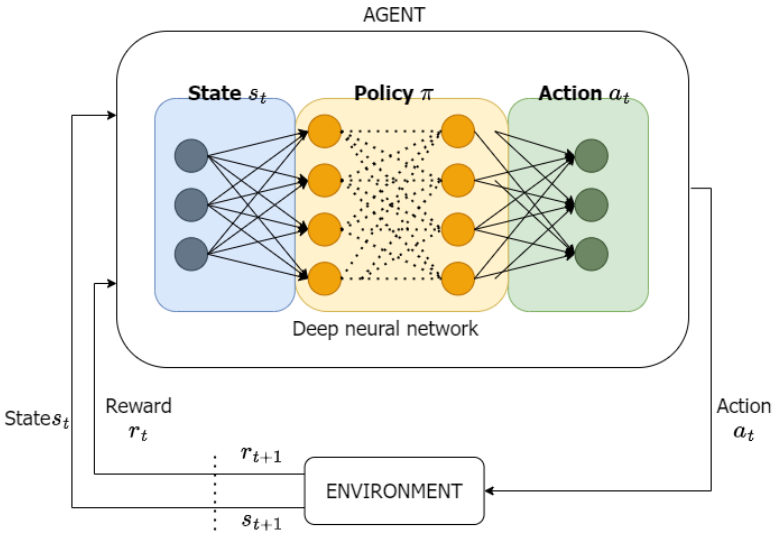


Figure 2.7: Deep Reinforcement learning illustration

low-dimensional representations or approximations of high-dimensional data[4]. Coupling a state-of-the-art convolutional neural network(CNN) with an agent that utilizes a live camera feed as input for environmental mapping could for instance let the agent view the environment as labeled objects instead of just pixel values, decreasing the perceived complexity of the agent's state space and allowing the agent to more easily and effectively learn the optimal policy or value function.
Additionally, deep neural networks can be used to represent the policy or value functions themselves, especially useful for reinforcement learning problems with continuous action and state spaces, requiring very complex policy or value functions, as deep neural networks have proven to be universal function approximators[29].

2.5.5 Actor-Critic reinforcement learning

Actor-Critic reinforcement learning algorithms are a subset of deep, model-free reinforcement learning techniques that use multiple deep neural networks [15] and combine the model-free approaches of policy optimization and Q-learning. Approaches in this category are characterized by the interaction between the actor, a deep neural network learning the policy function, similar to policy optimization, and the critic, a deep neural network learning the action-value function of the environment, similar to that of Q-learning. With deep neural networks, a loss function needs to be utilized to update the parameters of the networks. For the critic network $Q_\theta(s, a)$, the output is directly comparable with the reward received from the environment, and can directly be updated through the use of a loss function comparing the output with the received reward and apply-

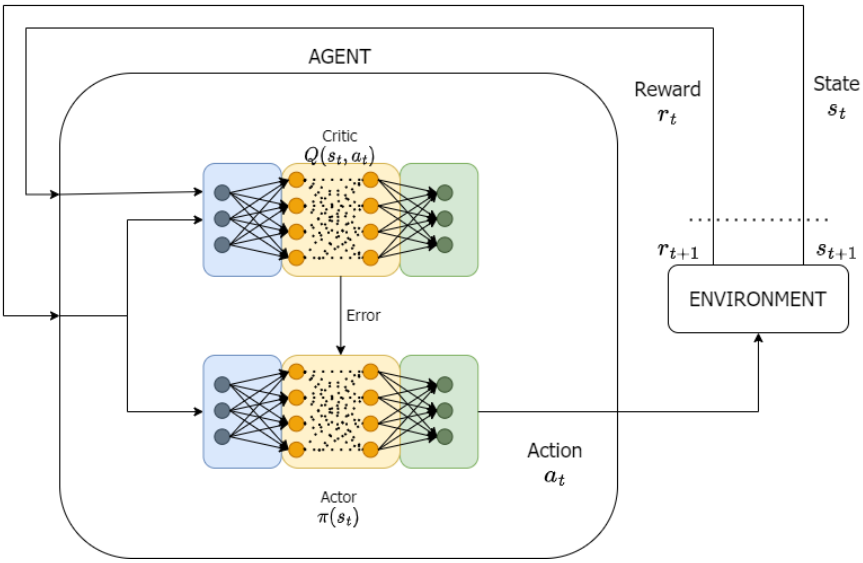


Figure 2.8: Actor-critic Reinforcement learning illustration

ing backpropagation. However, the actor-network $\pi_\theta(s)$ outputs an action a, which is not directly comparable to the reward received from the environment. Instead, the actor-network is updated by sampling the critic network to extract the expected return of choosing actions according to the current policy, which is then used to perform gradient ascent on the network's parameters to maximize the expected return.

2.5.6 Offline reinforcement learning

Offline reinforcement learning[28] is a data-driven version of reinforcement learning. Its main purpose is to allow reinforcement learning agents to be deployed in environments in which exploratory behaviors are deemed to be dangerous, which is the case for many robotic manipulation tasks because of the dangers of allowing a robotic manipulator to randomly move throughout the space it operates in. The object of the agent is still the same, for the agent to learn an optimal policy with a specific task or goal in mind. Learning is achieved by providing a static data set that the unit has to sample from instead of using an exploratory approach where the agent is allowed to interact with its environment freely to obtain experience in the domain. The agent must thus learn its optimal policy from the state transitions provided in the data set, which makes the approach more closely resemble standard supervised learning.
Negating exploration has its own set of problems. Since the agent has to entirely rely on the given data set, it will most likely not have access to the entire state transition space, and thus not contain action-transition pairs that are viable and good action choices for certain situations. Furthermore, it completely removes the agent's ability to learn when encountering novel

situations in which appropriate actions cannot be inferred from the given data set, making the method less resilient to changes in the environment. Some of the problems that offline reinforcement learning methods attempt to negate can also be partially negated by training the reinforcement learning agent in a simulated environment. Training in a simulated environment does however also pose its own set of problems when the agent is transferred to the real world.

2.5.7 Transfer learning

Transfer learning is a method of using previously attained knowledge from a specific task or environment to achieve accelerated learning speed and accuracy in a new task or environment, sufficiently similar to the task or environment the agent was originally trained for. The approach offers an alternative to offline reinforcement learning, as it negates dangerous exploratory behavior by having the agent sufficiently trained in a simulated environment before being deployed in the real world. Transfer learning seeks to overcome generalization problems, in which an agent becomes so specialized at high performance on a single task or in a single environment that its learned behavior cannot be utilized in similar tasks and in similar environments. Some strategies to mitigate this problem are mentioned in an article from 2018[12], the most relevant of which is lifelong learning and data augmentation when considering initially training a reinforcement learning agent for manipulator control in a simulated environment. In a simulated environment, it is possible to achieve good generalization through data augmentation, which is slightly altering the agent's input data in such a way that the agent and its trained networks do not overfit to the particular task. However, on tasks that require dexterous manipulation ability, a slight alteration in the input data may often cause an agent to fail in its task. Instead of altering the input data of the agent for each step in the reinforcement learning process, the initial conditions of the simulated experiment could be slightly altered each time an episode is instantiated, forcing the agent to learn the general solution to the task instead.
The other relevant form of transfer learning that is mentioned is lifelong learning, which is described as the capability of a system to learn multiple tasks over a lifetime in one or more domains. This is possible due to having multiple tasks sharing network parameters. The proposed way of achieving lifelong learning is training the agent sequentially, having it learn in one environment or one task at a time, then continue the learning process in another environment or on another task. However, the process can be problematic because of catastrophic forgetting, in which an update to the network's parameters overrides the previous weights in a manner that causes the agent to no longer be able to perform a specific task. An

approach for mitigating this is to include an experience replay memory for the agent, which is a memory of previously experienced interactions with the environment. The memory can be sampled from so that it no longer matters if important parameters in the network are overwritten by interaction in the new environment. The inclusion of such an experience replay memory is already mandatory for many reinforcement learning approaches, especially approaches that include value functions Q, such as actor-critic methods or deep Q-learning. The problem can also somewhat be mitigated by going back and retraining on previously learned or similar tasks to re-learn the parameters that were overridden, but this approach also risks catastrophic forgetting of the newly trained task.

2.5.8 Reinforcement learning in robotics

Some tasks, such as many robotic manipulation tasks, might seem intuitive because of the ease an average adult human has at performing them. However, they quickly become increasingly complex when having to consider how to sufficiently describe the problem, and design a reinforcement learning agent that is capable of solving it. The engineer has to take into account dynamic environments, human interaction, and other obstacles that real-world deployment may bring. One often used term for this increase in complexity and often hard transition from a simulated or theoretical environment to the real world is the 'reality gap'[23]. Most often used in the context of evolutionary robotics, in which the problem derives from the robot's controllers being evolved or generated in a simulated environment where parameters like gravitational force, surface tension, and other physical force simulations are only approximations of their real-world counterparts. For the same reasons, the term is also highly relevant in the field of reinforcement learning, especially when considering transfer learning from a simulated environment to a real-world manipulator.
In a paper from 2018[31], an attempt was made to benchmark some reinforcement learning algorithms' performances on specific tasks when deployed on real-world robotic manipulators. It was found that the performance of the deployed reinforcement learning agents was highly sensitive to their given hyper-parameters, such as learning rates, sample batch sizes, and update rates, which often required re-tuning when a new task was presented. However, some agents achieved effective learning for a wider range of hyper-parameters, showing that an agent could potentially learn multiple tasks using the same hyper-parameters as long as the ranges for effective learning for the respective tasks somewhat overlap.
In another article from 2021[22], the authors discuss how deep reinforcement learning agents can be deployed to a wider range of different robotic setups and the challenges of making them perform and learn efficiently

in the real world. The article highlights the fact that even though deep reinforcement learning is regarded by some as being too inefficient for real-world scenarios, advances made in the field have shown that it is in fact fully feasible for a reinforcement learning agent to learn complex tasks ranging from quadrupedal locomotion to dexterous manipulation, but doing so may require careful and deliberate design of the agents. .

2.6 Image classification and object detection

The focus of the project is not primarily image classification and object detection, but the procedures are essential components of TIAGo's perception and understanding of the environment it operates in. Thus the topics are only presented on a surface level to understand the specific object detection network utilized.

2.6.1 Convolutional layers

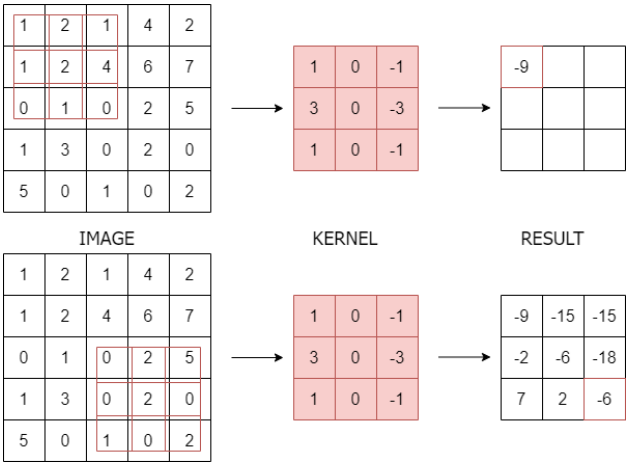


Figure 2.9: Image convolution example of a 5x5 image with 3x3 kernel, stride = 1, no padding.

In neural networks used for image classification or object detection, the most commonly used technique is convolutional layers. Image convolution is the mathematical operation of multiplying every pixel of an image with a fixed-size kernel or filter, represented by a matrix, that contains values that define a specific operation. The convolution is performed by sliding the kernel center over every pixel of the image and calculating the sum of all the kernels values multiplied by the pixel value it overlaps on the image, the sum is then placed at the pixel coordinate of the pixel the center of the kernel overlaps in the resulting image. The operation can be adjusted to only include pixel coordinates of the image in

which the kernel fully overlaps it, resulting in the output of the operation having a smaller height and width than the original image, as can be seen in fig. 2.9.
A single convolutional layer in a convolutional neural network(CNN) is made up of many kernels, performing the convolution operation on its input, replacing the nodes described in sec. 2.4. The resulting structure after a convolution operation has been applied is called a 'feature map' and thus the output of a convolutional layer is many feature maps. The values that fill the kernels of the layers are weights that are iteratively updated using backpropagation. In addition, a parameter called stride s can be included, causing the kernel to move s pixels along the image's axis between each calculation, skipping some positions and resulting in an even smaller output. An additional operation, called 'pooling' is also utilized, replacing a group of neighboring pixels or values of a feature map with a singular value, often chosen as either the minimum, the maximum, or the average value in the neighborhood, resulting in a downsampling of the image or feature map. Typically, each convolution layer iteratively shrinks the height and width dimensions of the preceding image or feature maps but introduces new feature detectors, resulting in a larger number of smaller feature maps. The last collection of feature maps is usually fed into a fully connected layer, allowing each feature map to represent a higher-level feature of the original image, which can be used to classify the whole image or identify objects in the image. An example of the process can be seen in the architecture of the YOLO object detection network in fig. 2.10.

2.6.2 YOLO (you only look once) object detection

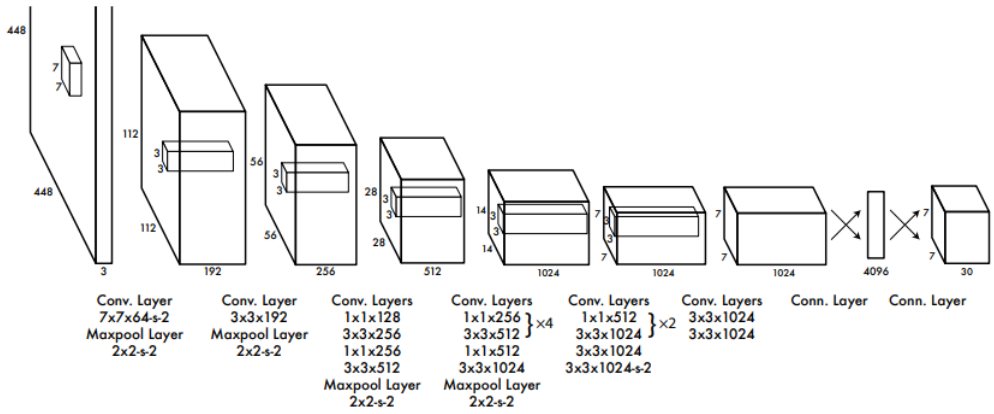


Figure 2.10: YOLO architecture. *From [37].*

YOLO [37] is a single-network, real-time, object-detecting network that has been adapted to be used with ROS [6]. Its architecture is composed of 24 sequential convolutional layers with some pooling layers in between,

followed by 2 fully connected layers. The output of the network is a number of bounding boxes, one for each detected object, along with their corresponding predicted object class and prediction confidence. The network is built with real-time detection in mind and supports framerates up to 45 frames per second. Since the architecture is mainly built for speed and robustness, the authors state that it can struggle with limitations such as detecting similar objects in groups, objects in new and unusual aspect ratios, and bounding box errors for small objects. The bounding box precision errors make the network unable to accurately be used to find the exact position of the detected objects alone. However, this issue can somewhat be mitigated by the utilization of TIAGo's depth camera, as presented later.

Chapter 3

Methods

3.1 Primary simulation environment

Both ROS and TIAGo's ROS implementation offer valuable tools that are utilized in this project. At the core, ROS's topic and messaging systems are largely useful as it allows for a modular approach when creating the scripts and programs utilized in the simulation of the robot. While most of ROS and TIAGo's modules are written in C++, a lot of modern and widely used machine learning libraries are written for Python. Some of the Python libraries required for effective reinforcement learning training are made for newer versions of Python, and will not directly work with packages and libraries written for older versions. Computer vision libraries, such as the openCV version utilized in TIAGo's software packages, will only work for older, deprecated versions of Python. However, even though the libraries and packages themselves are incompatible, scripts can concurrently be run, communicating with each other via ROS topics, through the use of ROS messages, overcoming software incompatibilities. This section briefly explains some of the key packages, and why they are relevant.

3.1.1 Actionlib

Actionlib is the library used in this project to control TIAGo's joints in the Gazebo simulation environment. It works similarly to a setpoint controller; for a timestep, set a goal for each joint in the arm, this could for instance be "rotate arm joint number 2 to π", and is called an action request. For the control of TIAGo's arm, a single request must contain seven values representing the angular displacement each of the arm's revolute joints should reach in the selected timestep. The action request is then sent and the robot will try to execute the action by interpolating between the current state of the joint or joints requested and the goal.

A result is returned that contains information about the execution of the action; it can be successful or a failure. A failed action could for instance be if the action request given described a motion that was too large for the given timestep to be properly executed, or it could be that a hindrance in the environment stopped the arm from reaching its goal. Checking the result is a component of how one of the reward functions described later is calculated.
By controlling the manipulator in this fashion, by choosing discrete 'goal' values for each controllable joint, the actual control function $q_n(t)$ for a joint n can be described by interpolating between the goal values. To approximate smooth movements, the timestep chosen for each action must be small to achieve a high rate of discrete control points. In fig. 3.1 the approximation of a sine function is visualized through the use of discrete control points and interpolation. On the left, the actionlib controller is called at a rate of 4 Hz, resulting in a reasonably bad approximation of the function, on the right, the actionlib controller is called at a rate of 10 Hz, resulting in a much-improved approximation.

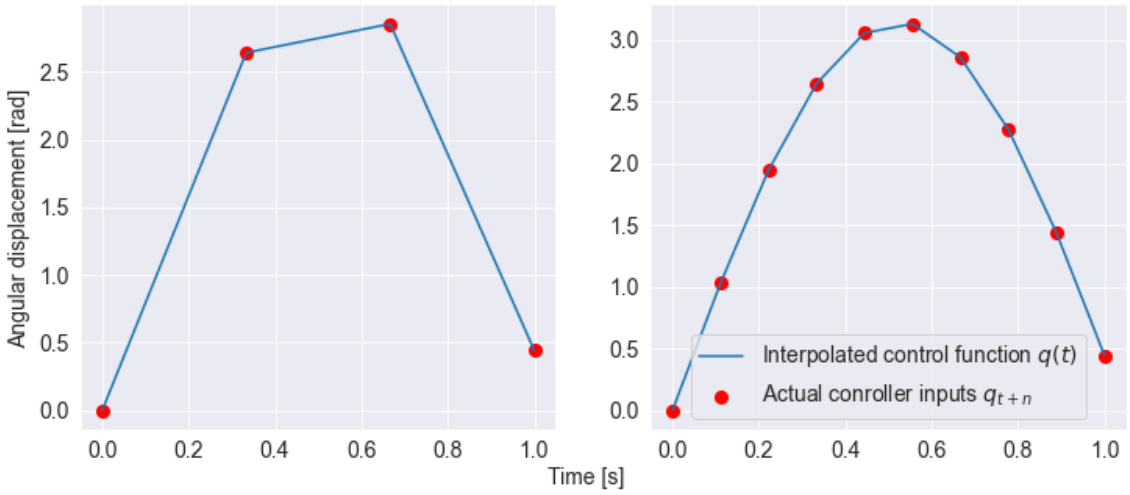


Figure 3.1: Actionlib control visualization

3.1.2 TF

Tf is the package that lets the agent keep track of all the moving and interlinked coordinate frames of the robot/manipulator. It provides real-time updates on the relationships between all the different coordinate frames and provides real-time updated homogenous transformations between them. In this project, the package is used to keep track of the position of TIAGo's arm in space, the spatial relationship between the end-effector and objects or hindrances in the robot's environment, and to transform

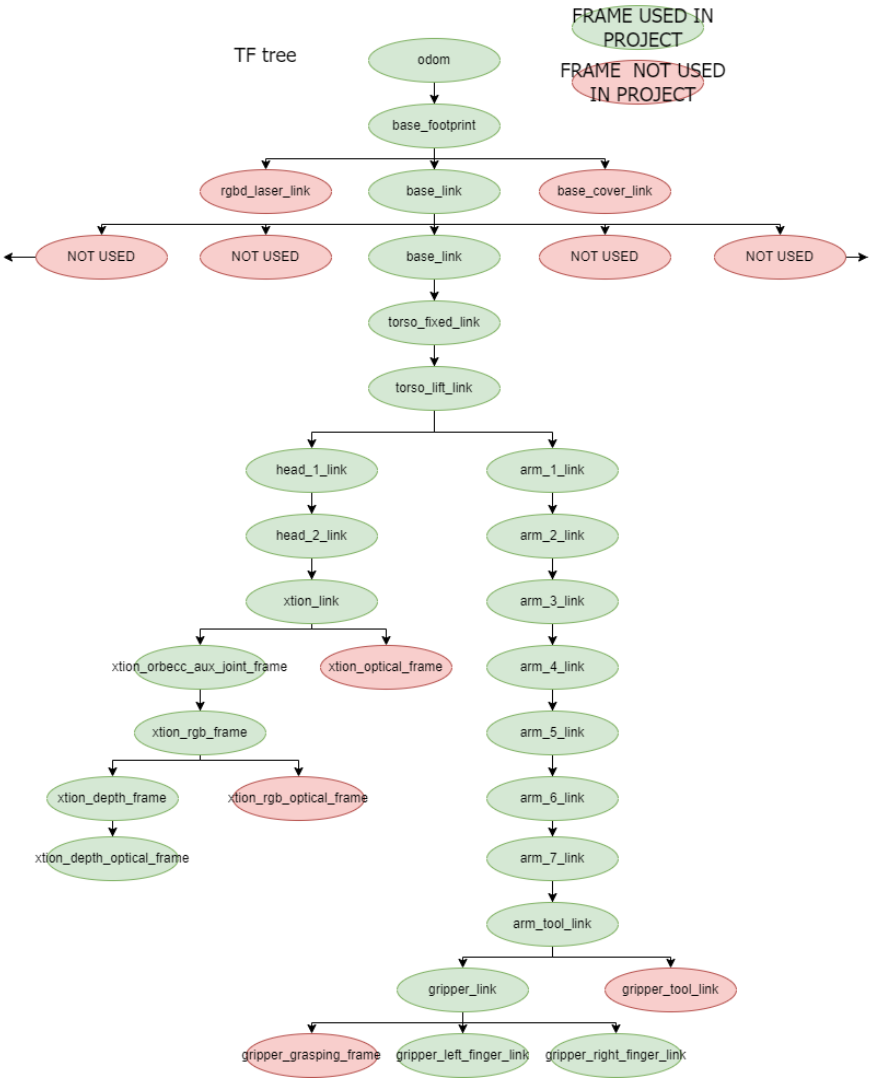


Figure 3.2: TF tree of TIAGo with only used frames highlighted

data recorded by TIAGo's sensory suite, such as data recorded by the RGB-D camera, to a common reference frame. In fig. 3.2, TIAGo's TF tree is provided. The frame called 'xtion_depth_optical_frame' contains the reference frame for the camera's depth information while the frame called 'gripper_link' contains the reference frame of the end-effector. The frame called 'base_footprint' is used as the common reference frame, and referred to later as the 'base coordinate frame'.

3.1.3 OpenCV

With cv_bridge it is possible to fully integrate OpenCV into ROS. It is used to decipher and utilize the data recorded by the camera of the robot. It is a core package used in image-related tasks and is thus also a core com-

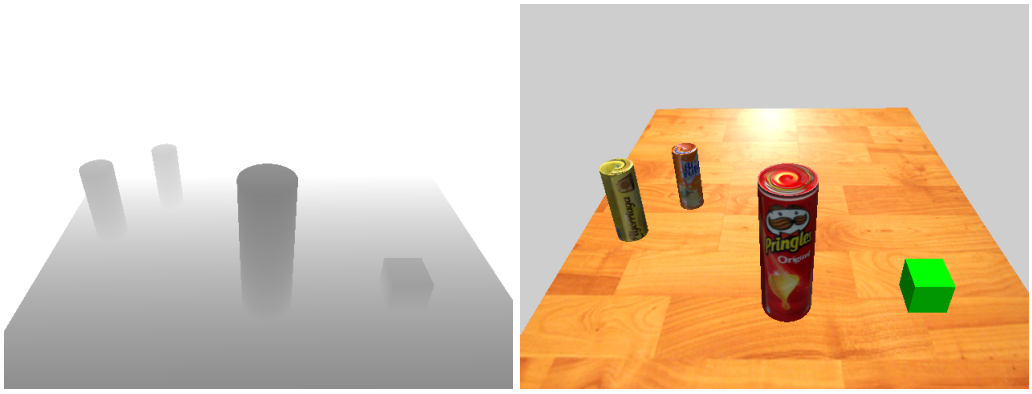


Figure 3.3: Depth image and RGB image captured by TIAGos camera, viewed through OpenCV.

ponent in object detection. Both YOLO object detection and point cloud utilization are dependent on this package. TIAGo's image streams can be seen in fig. 3.3.

3.2 Secondary simulation environment

In addition to the simulation environment provided in the ROS implementation and software packages of TIAGo, implemented in Gazebo, a secondary simulation environment is also utilized. The secondary simulation environment is implemented in the PyBullet simulation engine [9], and is implemented to allow for rapid testing of different tunable parameters and prototyping of different definitions of the state space, action space, and reward function for the reinforcement learning agents trained. The PyBullet simulation environment allows for parallelism, allowing multiple agents to train concurrently using the same reinforcement learning algorithm and policy. Additionally, it is easy to speed up, allowing for environmental steps to be made as soon as the hardware running the simulation allows it. The downside of this simulation environment is that in its current implementation, only proprioceptive sensory systems can be accurately simulated. This heavily limits the usage of TIAGo's full capabilities as it does not allow for its full sensory suite to be accurately simulated.

3.3 Reinforcement learning setup

In this section, the designs of the reinforcement learning agents are presented. The design of some of the reinforcement learning agents' components varies depending on which simulation environment is utilized for training

them, thus subsections concerning the state spaces and reward functions are split in two to allow for explanations of how and why the different components are designed as they are, and in which simulation environments the designs are applicable.

3.3.1 Inspiration

At the beginning of the planning process, a few papers on manipulator control through the use of reinforcement learning were selected as sources of inspiration for the implementation. Of those, a paper on control of a 7-DOF manipulator, somewhat similar to TIAGo's arm, through the use of actor-critic reinforcement learning methods [7] was sourced to determine which reinforcement learning algorithms to attempt to implement for TIAGo. The authors of the paper present actor-critic methods, such as deep deterministic policy gradient (DDPG), first introduced in [30] and soft actor-critic (SAC), first introduced in [18] as viable methods of learning manipulator control. However, in their paper, dexterous control of the gripper is not included, and they do not attempt to control the manipulator directly in its configuration space C, opting for the use of an Inverse kinematics (IK) solver to determine the configuration space parameters necessary to control the robot while the trained agents determined operating space positions for the end-effector of the manipulator to reach. In a sense, the agents trained in the project do not learn to directly control the manipulator, but rather generate a path that the IK-solver is responsible for following. While their results in themselves do not explicitly show promise for dexterous manipulation using actor-critic methods, the authors of the soft actor-critic (SAC) method have presented a paper[17] in which they train dexterous manipulation skills and legged locomotion by allowing the agent to directly control the robot in robotic setups less similar to TIAGo's arm.

3.3.2 The state space

In reinforcement learning the state space or observation space represents the available information that the agent has to determine its actions. Depending on what behavior is required from the agent and what environment it should operate in, the state space needs to be designed to contain enough information for learning the intended behavior while also reducing noise by restricting the available information in such a way that the agent does not get the information it does not necessarily need to determine its actions. Two state space schemes are presented, one of which is designed to utilize the full sensory suite of TIAGo and for the same reason are restricted to only be simulated in Gazebo. The other is more loosely designed and does not utilize TIAGo's full sensory

capabilities, but is able to be used in simulation environments in which TIAGo's exteroceptive sensory suite is not implemented.
For the first part of this project, a continuous state space that contains the distance of every joint in Tiago's manipulator arm to the closest environmental obstacle is used. The state space scheme is designed with simulation in Gazebo in mind and requires the simulation environment to be able to accurately simulate exteroceptive sensory components of TIAGo. The position of the joints in space is calculated through kinematic transformations from each joint's respective coordinate frame to the base frame of the robot and the position of environmental elements is found by transforming the point-cloud depth image generated by Tiago's RGB-D camera, from the camera's coordinate frame and to the base frame of the robot. This results in having a point cloud, containing environmental obstacles and points of each of the joints of the manipulator's arm in the same coordinate frame. A vector describing the distance and direction from each joint to the closest obstacle, found in the point cloud, can be found by calculating the Euclidean distance between the joint and all the points in the cloud and selecting the point p_{PC} in the point cloud that minimizes the distance and calculating $v = p_{PC} - p_{joint}$, in which p_{joint} is the center of the joint.
To incentivize learning to reach a target object, the state space must also contain the distance and relative position of the target object in relation to the end-effector. As long as the position of the target object is available, either through the use of pose and position approximation utilizing TIAGo's camera, or gathered directly from the simulated environment, the relative position and direction of the object in relation to the end-effector of TIAGo's arm can be calculated in the same manner as the environmental obstacles in relation to TIAGo's joints.
With this information in mind, the state space is represented as a matrix:

$$S_1 = \begin{bmatrix} d_0 & x_0 & y_0 & z_0 \\ d_1 & x_1 & y_1 & z_1 \\ d_2 & x_2 & y_2 & z_2 \\ d_3 & x_3 & y_3 & z_3 \\ d_4 & x_4 & y_4 & z_4 \\ d_5 & x_5 & y_5 & z_5 \\ d_6 & x_6 & y_6 & z_6 \\ d_7 & x_7 & y_7 & z_7 \end{bmatrix}$$

Where d_0 denotes the distance to the target object, $[x_0\ y_0\ z_0]^T$ denotes the vector from the end-effector to the target object, $d_n, n \in \{1, ..., 7\}$ denotes the distance from the manipulator's arm to the closest environmental obstacle, and $[x_n\ y_n\ z_n], n \in \{1, ..., 7\}$ denotes the vector from each respective joint in the manipulator's arm to the closest environmental obstacle of the respective joint.

The second state space scheme utilized in this project is designed to only concern the position of the end-effector in relation to the target object and does not include information about environmental obstacles. It is designed to only rely on proprioceptive sensory data and information from the simulation environment itself, allowing it to be utilized in both simulation environments:

$$S_2 = \{q, \dot{q}, p_{ee}, \bar{p}_{obj}\}$$

The state space contains $q \in \mathbb{R}^7$, the angular displacement of each of TIAGo's links, $\dot{q} \in \mathbb{R}^7$, the angular velocity of each of TIAGo's links, $P_{ee} \in \mathbb{R}^3$, a vector containing the position of the midpoint between TIAGo's grippers in the coordinate frame of the robot's base and $\bar{p}_{obj} \in \mathbb{R}^3$, a vector describing the relative position of the target object in relation to P_{ee} in the coordinate frame of the robot's base. The state space can also be extended to include two additional parameters θ and α, concerning the direction of the end-effector in relation to the target object, described later in sec. 3.3.5.

3.3.3 The action space

The action space is designed in an attempt to leverage some of the already implemented modules for robotics control in ROS and TIAGo. Using the ROS library called 'actionlib', described earlier, the agent chooses a goal joint value for each joint in the robot's arm, which the controller will attempt to reach in a defined time. The allotted time is decided by the rate at which the agent is called. For instance, a rate of 4 Hz would require the goal joint values chosen by the agent to be reached within a quarter of a second. The action space is thus defined as:

$$A = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \end{bmatrix}$$

Where $q_n, n \in \{1, ..., 7\}$ is the desired joint space configuration of joint n at the end of a single action. The values of q_n are constrained by the permissible range of motion of each joint and cannot have values outside of them (see appendix C). This choice of action space and action execution opens the possibility for very large and sudden movements. However, these movements can be disincentivized through the use of punishments in the reward function or limited by redefining the interaction between the action space and the controller.

3.3.4 Action space constraining

In [16], SAC is used to teach a four-legged mobile robot to walk. In that endeavor, it is stated that constraining the actions of the RL agent in some manner for the first n episodes is useful to prevent the agent from performing explosive and jerky actions at the beginning. This concept can loosely be extended to the manipulator arm of this project. As presented in the last section, giving the agent the ability to choose any next configuration q_{t+1} at time t from any configuration q_t, will result in explosive movements when actions are chosen at random. The agent can learn to restrict itself if incentivized to do so. However, the exploration done by the agent at the beginning of training requires random actions to be chosen and will mostly consist of movements that are too explosive in nature and either knock the robot down or displace it. The agent is thus left with a replay memory in which the overwhelming majority of state transitions are useless in terms of learning to reach a desired point in space with the end-effector of the arm. When these samples are fetched in mini-batches to be used to update the neural networks responsible for controlling the agent, the batches are unlikely to contain samples that are useful for learning the intended behavior, to reach for an object. Instead, the best sample in a batch will likely be a sample in which the robot is not knocked over or displaced, but also not moving toward the goal object in a meaningful manner. As a result, the agent learns to control the arm in a manner that avoids knocking over or displacing the robot but fails in learning to reach for the target object.
To mitigate this issue a simple change can be made to the action space and how the controller interacts with it. Instead of letting the agent pick any action q from the arms configuration space C, it is instead necessary to constrain each action. Let q_{MIN} and q_{MAX} be vectors containing the minimum and maximum angular displacements of each joint of the arm. $q_{DIFF} = |q_{MIN} - q_{MAX}|$ is then the maximum amount of angular displacement the agent can possibly perform in a single move. The interaction between the action and the controller can be redefined. Instead of letting $q_{t+1} = a,\ a \in A$, the next configuration is picked as $q_{t+1} = q_t + a,\ a \in A$, where $A = w_a[-q_{DIFF}, q_{DIFF}]$ and $w_a \in [0,1]$, named the 'action space constraining coefficient', is a scalar weight defining how far away from the current configuration q_t the next configuration q_{t+1} can be chosen from in the configuration space C. This allows the explosiveness of the moves to be manually tuned, and with a fairly low w_a, forces the agent to pick actions that lead to the next configuration q_{t+1} being fairly near the current configuration q_t in the configuration space C.

3.3.5 The reward mechanism

Designing a functional reward mechanism for the reinforcement learning agent requires leveraging the available information of each planned step in the robotic arm's movement and finding a way to give constructive feedback to the reinforcement learning loop. Finding an appropriate reward function is case-dependent and as such, there is no other way to design a good reward function than to take inspiration from other successful projects, iterate over ideas, and attempt to implement incentives for the agent to learn the behavior that is intended. Additionally, the reward function has to rely on information that is available in the state space of the agent, so that the changes in the received reward given by performing an action can directly be related to changes in the state space. In this project, different reward functions are used depending on what simulation environment the agent is trained in and what qualities that environment can measure.

Gazebo simulation environment

The first reward function discussed is primarily made for training and simulation in Gazebo. It is comprised of three parts, each with its own intention.

$$r_1 = -\frac{1}{2}d^2$$

The first part, r_1 is designed to encourage the agent to move the arm to a desired point in space. d is the distance from the end-effector to the position of the target object.

$$r_2 = -C$$

The second part, r_2 is used to check if any part of the arm is close to or colliding with the environment, represented by the point cloud of Tiago's RGB-D camera. C is a boolean, True if any part of the arm is close to or colliding with the environment.

$$r_3 = -S$$

The third and final part, r_3 is used to check if the action given to the joint controller is close enough to its current joint configuration to be completed within the allotted time frame. S, for step completion, is a boolean, True if the action given to the controller was fully completed within the set step time. The inclusion of the r_3 term ensures that the RL agent learns to constrain the movement given to the controller to a range that is physically and safely possible to do in a single timestep.

$$R = w_1 r_1 + w_2 r_2 + w_3 r_3$$

The total reward is the sum of each of the reward function's parts multiplied by tunable scaling factors w_1, w_2, w_3, which dictates how much each part is weighted against the others.

PyBullet simulation environment

The reward functions designed to work with the PyBullet environment are designed to be easily transferable to Gazebo. In this environment, the RGB-D camera of the robot is not implemented and it is thus impossible to simulate collision detection in a meaningful way that would be transferable to the real robot without further integration of perception modules. Instead, only the internal and measurable state of the robotic arm is used. The position of the object, which in this case is gathered directly from the simulated environment, can also easily be derived by the camera of the robot. Two versions of this reward function are presented, one of which only considers the position and state of the robotic arm, while the other also considers the rotation of the arm's end-effector in relation to the target object.

$$r_1 = \frac{d_t - d_{t+1}}{d_t}$$

Where d_t is the distance from the end-effector to the object before the action a at time t is executed, and d_{t+1} is the distance from the end-effector to the object after the action is executed. r_1 thus measures the percentage of the distance before the action is executed that the chosen action is able to close. Using the percentage of the distance that the action is able to close, rather than the distance itself, makes the reward scale better with distance, giving equal reward and weighting for large movements when the arm is far from the target object and small, precise movements when the end-effector is approaching the target object.

$$r_2 = -|q_{t+1} - (q_t + a_t)|$$

r_2 measures the error of the executed action, giving a penalty for choosing actions that are too large to be executed within the allotted time frame of the step. The error definition requires actions to be chosen in the manner presented in sec. 3.3.4.
The total error of the reward function only considering the position of the end-effector in relation to the target object is given by:

$$R = w_1 r_2 + w_2 r_2$$

Where w_1, w_2 are scalar weight parameters to tune the different parts of the reward function in relation to each other. R thus only considers the distance of the end-effector in relation to the object but also puts some weight on learning to pick actions that are achievable within the allotted

time frame of each step.

Another reward term can be added to motivate the agent to control the end-effector's direction in relation to the object. Doing so first requires a few new parameters to be introduced. Why this is necessary is visualized in fig. 3.7.

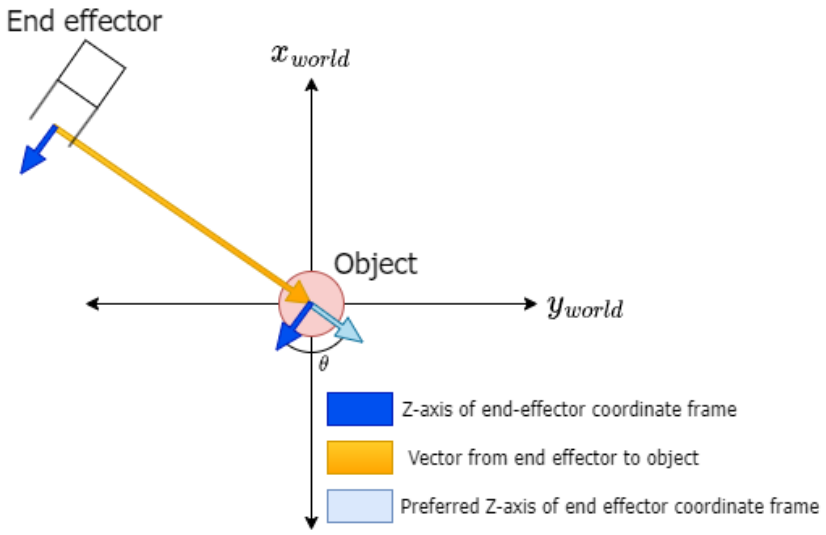


Figure 3.4: Inferring the preferred direction of the Z-axis of the end-effector's coordinate frame.

To find a suitable way to calculate a reward for the direction of the end-effector in relation to the target object, preferred directions for two of the coordinate frame axis of the end-effector have to be inferred. First, the preferred direction of the end-effector's Z-axis has to be defined. Since information about the target object's position in relation to the end-effector already exists in the state space, a preferred direction can easily be inferred. Ignoring the elevation of the end-effector (Z-axis of the world), the vector from the end-effector to the target object can be projected down on the XY-plane of the world. Doing so allows the preferred Z-axis of the end-effector's coordinate frame to stay perpendicular to the Z-axis of the object, and thus negates the end-effector hitting the object at a steep angle if the preferred direction is followed. See fig. 3.4. The difference between the unit vector representing the direction of the Z-axis of the end-effector's coordinate frame Z_{ee}, given in the base coordinate frame, and the preferred Z-axis of the end-effector Z_{pref}, also given in the base coordinate frame, is

$\theta \in [-\pi, \pi]$, and can be found with:

$$\theta = \arccos\left(\frac{Z_{ee} * Z_{pref}}{|Z_{ee}| * |Z_{pref}|}\right)$$

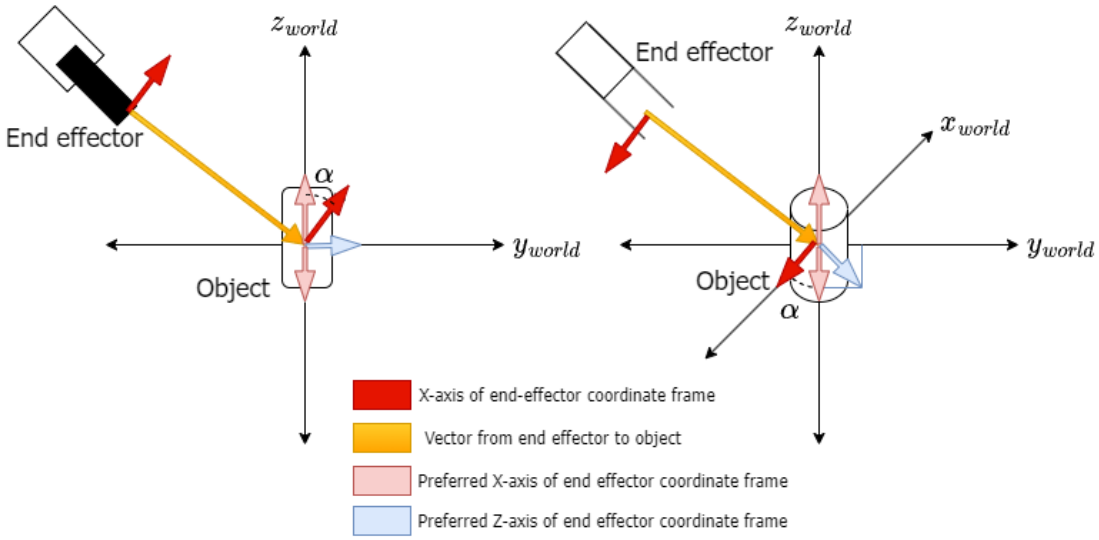


Figure 3.5: Inferring the preferred direction of the X-axis of the end-effector's coordinate frame.

The second part of calculating the reward incentive involves inferring a preferred direction of the X-axis of the end-effectors coordinate frame. Since the preferred Z-axis described earlier lies on the XY-plane of the world, it will always be perpendicular to the Z-axis of the world. Thus if the X-axis of the end-effector's coordinate frame is parallel to the Z-axis of the world, and the Z-axis of the end-effector is pointing directly at the object on the XY-plane of the world, by following the preferred direction of its Z-axis, the grippers will be in a position to wrap around the abject and not collide with it. α is thus defined as the angular difference between the unit vector describing the X-axis of the end-effector's coordinate frame and the Z-axis of the world. It is worth noting that it does not matter if the X-axis of the end-effector's coordinate frame points upward along the Z-axis of the world, or downward, as both allow the end-effector's grippers to be rotated such that they can wrap around the object. Thus $\alpha \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ See fig. 3.5.

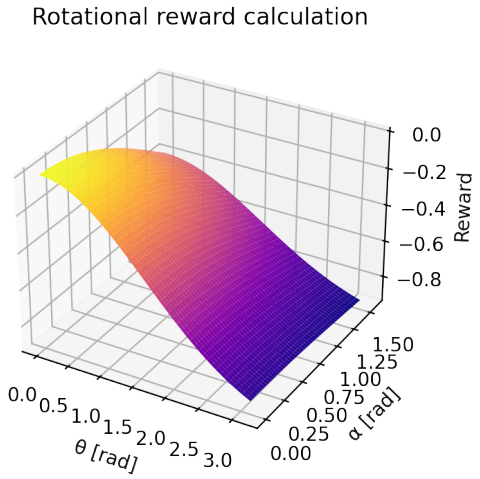


Figure 3.6: Reward based on direction and rotation of end-effector.

With both θ and α defined, an incentive has to be added to the reward function, such that the agent is incentivized to minimize them. For this purpose, a two-dimensional Gaussian is fitting, see fig. 3.6:

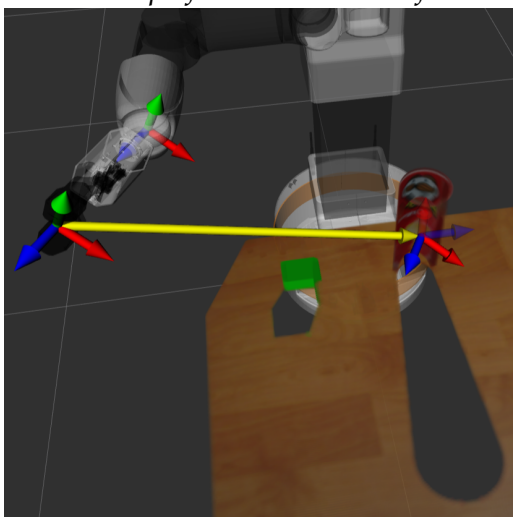
$$r_3(\theta, \alpha) = \exp\left(-\left(\frac{\theta^2}{2(\frac{\pi}{2})^2} + \frac{\alpha^2}{2(\frac{\pi}{2})^2}\right)\right) - 1$$

Using a Gaussian results in the added punishment for not following the preferred directions being in the $[-1, 0]$ range, allowing for easier tuning of the weight associated with it. This change results in the reward function being:

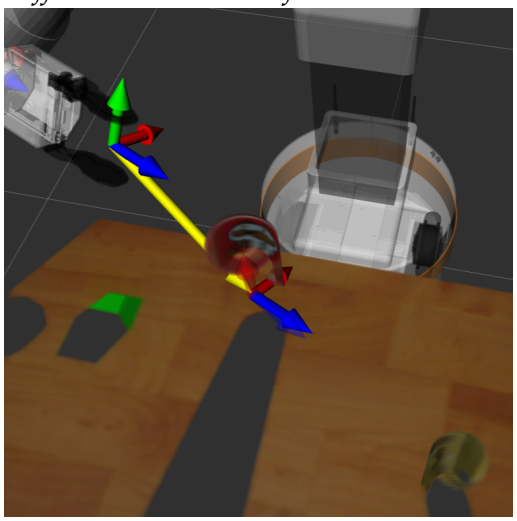
$$R = w_1 r_1 + w_2 r_2 + w_3 r_3$$

Where w_n and r_n, $n \in \{1, 2\}$ are the same weightings and terms described earlier, r_3 is the new term concerning the direction of the end-effector and w_3 is its weighting.

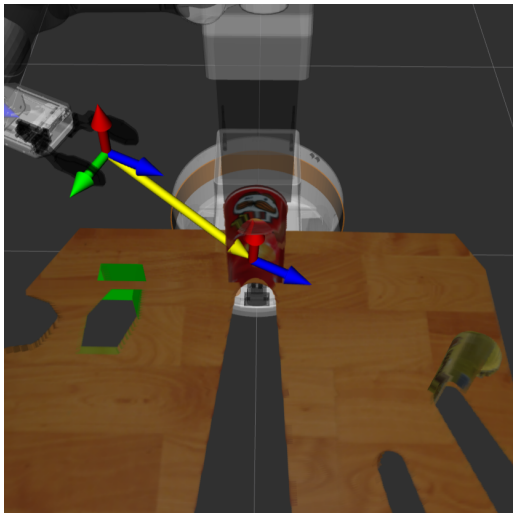
Examples of end-effector directions, color-coded vectors are as follows; red is the direction of the x-axis in the respective coordinate frame, blue is the direction of the z-axis in the respective coordinate frame, green is the direction of the y-axis in the respective coordinate frame and yellow is the vector from the end-effector to the target object. The semi-transparent blue and red vectors represent the preferred directions of the end-effector's coordinate frame.



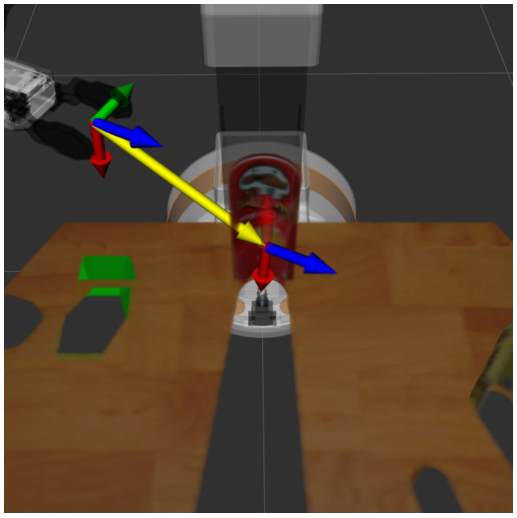
Example A: Bad direction, the end-effector is pointing away from the object.



Example B: Bad direction, the end-effector is pointing towards the object but the grippers are not lined up to wrap around it.



Example C: Good direction, the end-effector is pointed towards the object, and the grippers are lined up to wrap around the object.



Example D: Good direction, same as *Example C*, except the end-effector is rotated π radians.

Figure 3.7: Gripper direction examples

3.3.6 Reward scaling

The importance of properly scaling the reward function for efficient learning is discussed in [16]. It is found that for SAC, most of the benchmark reinforcement learning problems attempted by the authors were most efficiently solved with a reward scale of 0-100, implying that the reward function should be adjusted to provide the agent with rewards in that range to optimize learning. However, this finding was not absolute, and some problems required the reward function to be tuned to provide rewards in orders of magnitudes higher. Reward scaling is thus case-dependent, but should be designed to provide rewards in a clearly defined range. The weights of the reward functions presented in the previous section should thus be tuned by this principle.

3.3.7 The state-transition memory

When doing model-free deep reinforcement learning with continuous action and state spaces it is vital for the agent to be able to build some sort of notion of how the environment operates. Since both the action space and state spaces are continuous, a discrete structure such as a table cannot be utilized. Instead, a state-transition memory or replay memory is built. In most algorithms utilizing a replay memory, each state transition is stored in the replay memory D as a tuple:

$$e_t = (s_t, a_a, r_t, s_{t+1})$$

Where each experience e at time t is stored as the state s_t at time t when an action was taken, the action at the time a_t, the reward that was received by performing the action r_t and the state of the environment after the action was performed s_{t+1}.
The replay memory is sampled from when updating the deep neural networks implemented in the reinforcement learning agent. The idea behind using a replay memory instead of updating sequentially is that if the networks are updated sequentially and the experiences e_t are discarded as the agent trains, each subsequent experience would be highly correlated with the last, which would lead to inefficient training. Instead, the memory unit is built upon and randomly sampled from when updating the networks. In this fashion, a batch of samples randomly selected from the memory should contain experiences that are not as correlated and thus lead to more efficient training. The replay memory can also play a role in transferring the agent from one environment to another, as described in sec. 2.5.7.

3.3.8 The optimizer

Since the reinforcement learning agent contains deep neural networks, a way to update and maintain these networks has to be implemented. To do so, an optimization algorithm is utilized. The goal of the optimization algorithm is to update the parameters of the network as efficiently as possible. For this task, ADAM, first introduced in [25] is used. The two features that make ADAM a better choice than other optimization algorithms are adaptive momentum and adaptive learning rates. Firstly, when updating the network parameters, ADAM does not only use the currently calculated gradient but also an exponentially decaying average of the past gradients. This feature allows past gradients to contribute and 'push' network parameters out of local minima or optima when the current gradients become small. Secondly, since the exponentially decaying average of the past gradients is stored, it is also utilized to perform parameter-wise updates to the learning rate of every network parameter. This allows learning rates in parameters with few updates and many zero gradients to be adaptively turned up and learning rates for volatile parameters that are changing too quickly to be tuned down.

3.3.9 Neural network structure

Different reinforcement learning problems can often require different neural network structures, even when using the same reinforcement learning method. The structures of the networks utilized in training the agents in this project are set by the authors of the actor-critic methods used for training. Both primary algorithms use similar network depths but use slightly different parameters in their networks. For SAC [17], it is stated that using two hidden layers with a size of 256 each, with RelU works well for some robotics problems. For DDPG [30] it is stated that the authors used two hidden layers with ReLu activations, with sizes 400 and 300 respectively for direct input problems, and three convolutional layers + 2x200 hidden layers for camera input problems. Other publications using these methods, such as [7], do not explicitly state their network structures and are thus assumed to use the same structures and hyperparameters as presented in the original papers.

3.3.10 Rate of control

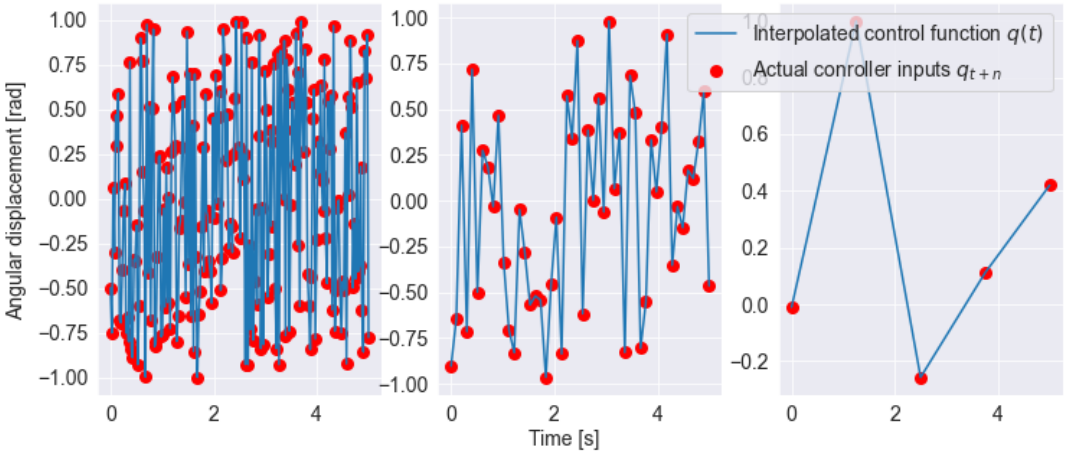


Figure 3.8: The effect of rate when the agent is exploring. **Left:** The next configuration is sampled at a rate of 20 Hz. **Middle:** The next configuration is sampled at a rate of 10 Hz. **Right:** The next configuration is sampled at a rate of $\frac{1}{2}$ Hz.

As described in sec. 3.1.1, the actionlib controller controlling TIAGo should be called at a defined rate, requiring another hyperparameter to be tuned. In the section, it is described how a high rate is required to approximate smooth movements when controlling the manipulator. However, smooth control might not be a goal in itself. The chosen rate has to balance the interaction between discrete state transitions, the reward functions' response to the state transitions, and the smoothness of movement. Having a high rate will not in itself guarantee smoother movements when the movements are controlled by a reinforcement learning agent, and will most likely induce shakiness during the exploration process at the beginning of training when movements are completely random, caused by many quick subsequent changes in direction. On the other hand, a rate that is too low might allow the agent to move the arm to any place in its operating space in a single time step, not allowing for any adjustments to be made while moving. Thus a rate has to be picked such that the agent is able to adjust its movement at a rate that allows it to react to the environment but also experience state transitions that are meaningful for learning how to behave in the environment. In fig. 3.8, three different rates of control are visualized when the next state is chosen at random for values in the range $[-1, 1]$ over 10 seconds. To the left, a rate of 20 Hz is shown, showing rapid changes in the direction of the control function, possibly inducing shakiness and instability to the manipulator, producing samples that are not optimal for training. In the middle, a rate of 5 Hz is shown,

showing some rapid changes in the direction of the control function, but also showing a few, more controlled state transitions. To the right, a rate of $\frac{1}{2}$ Hz is shown, showing very slow and controlled transitions between states, but exploring fewer states during the 10 seconds.

3.4 Algorithms

Two actor-critic algorithms are tested in this project, one is stochastic and one is deterministic. In practice, this means that the output of the policy neural network in the agent is interpreted differently. With a deterministic policy, an action a_t at time t is deterministically chosen as a function $\mu_\theta(s_t)$, given the neural network weights θ and current state s_t. In a fully trained agent where no noise is added for exploration, the chosen action is always the same for a state s_t. On the other hand, with a stochastic policy, an action is chosen from a probability density function. In this case, the output of the policy network of the agent is not an action itself, but rather the mean actions $\mu_\theta(s_t)$ and log standard deviations $\log \sigma_\theta(s_t)$, that are used to build the probability density functions an action is sampled from. Action sampling from a distribution is done as $a_t = \mu_\theta(s_t) + \sigma_\theta(s_t) \odot z$, where $\odot$ denotes element-wise multiplication and z represents a vector of noise from a spherical Gaussian ($z \sim N(0,1)$). Notation wise, $a_t = \mu_\theta(s)$ means an action is chosen deterministically and $a_t \sim \pi_\theta(\cdot|s)$ means an action is sampled from a probability density function.

3.4.1 DDPG

Deep deterministic policy gradient(DDPG)[30] is an actor-critic, model-free algorithm. It is based on a deterministic policy, and thus noise is added to the deterministic action decision in the early stages of training to provide for some exploration. Because of the deterministic nature of DDPG, it might struggle when determining actions for states that slightly deviate from what it has previously encountered.
The goal of the algorithm is to learn the optimal action-value function $Q^*(s,a)$ and optimal policy $\mu^*(s)$ through the use of deep neural networks $Q(s,a|\theta^Q)$ and $\mu(s|\theta^\mu)$, with parameters θ^Q and θ^μ. The parameters of the action-value network θ^Q are updated by the mean squared bellman-loss (MSBE) of the network, derived from the bellman equation for the optimal action-value function, described in sec. 2.5.3, and found by sampling previously experienced state transitions sampled from a replay memory D:

$$L(\theta^Q, D) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim D}[(Q(s,a) - (r + \gamma \max_{a'} Q(s',a')))^2]$$

In which $(s, a, r, s') \sim D$ indicates that the state s, action a, reward r and next state s' is sampled from the replay memory D and that γ is the discount factor. However, the current state of the loss function is unstable because the target $r + \gamma \max_{a'} Q_{\theta^Q}(s', a'))$ depends on the same parameters θ^Q as the network itself. Instead, a trick is utilized. By also including a target network Q', with parameters $\theta^{Q'}$, initialized to be equal to the parameters of the actual action-value network Q, and updating it by setting the parameters $\theta^{Q'} = \tau\theta^{Q'} + (1 - \tau)\theta^Q$ with $\tau \in [0, 1]$ each time the actual network is updated by backpropagation. The update method results in the target action-value network always lagging behind the actual action-value network. With the same trick also applied to the policy network μ, creating a target policy network μ', that also lags behind the actual network, the loss function can be redefined to be:

$$L(\theta^Q, D) = \mathop{\mathbb{E}}_{(s,a,r,s')\sim D}[(Q(s, a) - (r + \gamma Q'(s', \mu'(s')))^2]$$

In which the action-value network Q is updated by calculating the target from the target networks instead of itself.
The policy network μ is updated by performing gradient ascent on the network's parameters θ^μ, with respect to the action-value network Q by calculating the action-value estimations $Q_{\theta^Q}(s, \mu_{\theta^\mu}(s))$ for actions a produced by the sampled states s from the replay memory, using the current policy. The algorithm is described in alg. 1.

3.4.2 SAC

Soft actor-critic(SAC)[18] is a stochastic actor-critic, model-free reinforcement learning approach based on maximizing entropy while training. This is done by including an entropy and temperature parameter in the policy definition. In practice, it means that the goal of the algorithm is to train the agent to reach the goal in an entropic manner, resulting in a policy that is trained to more robustly handle slight deviations from what it has previously been trained on. It also utilizes two action-value functions instead of one, as described in [13]. This trick is shown to reduce value function overestimation which commonly occurs when using only one action-value function.
The goal of the algorithm is to learn the optimal action-value function $Q^*(s, a)$ and optimal stochastic policy $\pi^*(s)$ through the use of deep neural action-value networks $Q_1(s, a|\theta^{Q_1})$, $Q_2(s, a|\theta^{Q_2})$, with parameters θ^{Q_1}, θ^{Q_1}, and the stochastic policy network $\pi(s|\theta^\pi)$ with parameters θ^π. Target networks $Q'_1(s, a|\theta^{Q'_1})$ and $Q'_2(s, a|\theta^{Q'_2})$ are also utilized and updated in the same manner as the target networks used in DDPG. The network update process is very similar to DDPG, except for the inclusion of the entropy term and the utilization of two action-value networks instead

of one. The entropy term $H(\pi_{\theta^\pi}(\cdot|s)) = -\log \pi_{\theta^\pi}(a|s)$ denotes the entropy of an action a given the probability density function produced by the stochastic policy network $\pi(s|\theta^\pi)$ for a state s. The agent gets the calculated entropy of the action a given the policy π as a bonus in addition to the reward received due to the reward function. This, in turn, causes the definition of the loss function used in DDPG to be somewhat changed. Instead of calculating the targets as $r + \gamma Q'(s', \mu'_{\theta^{\mu'}}(s'))$, as presented in DDPG, the targets in SAC are computed as:

$$r + \gamma(\min_{j=1,2} Q'_j(s', \tilde{a}') - \alpha \log \pi(\tilde{a}'|s')),\ \tilde{a}' \sim \pi(\cdot|s')$$

In which $\min_{j=1,2} Q'_j(s', \tilde{a}')$ denotes that the lowest value produced by the target action-value networks are used, $\alpha \log \pi(\tilde{a}'|s')$ denotes that the entropy term is included, with a parameter α which determines how much the entropy term should weight in the calculation, and $\tilde{a}' \sim \pi(\cdot|s')$ denoting that the action $\tilde{a}'$ is not experienced during training, but rather sampled from the current policy network. The gradient ascent performed to update the policy function is also altered to include the entropy term and utilize the minimum of the two action-value functions instead of just a single one. The algorithm is described in alg. 2.

3.4.3 Algorithm comparison

The two actor-critic reinforcement learning algorithms that are utilized to train the agents used in this project, SAC and DDPG, are very similar. The main difference between them is how the policy is defined and the inclusion of the entropy term. It is stated [18] that SAC is more sample efficient and less prone to be heavily dependent on hyperparameter tuning, in other words, more robust. This can be explained by the way actions are picked by the trained policy, since SAC is more entropic and depends on a stochastic policy definition, a SAC agent executes a larger variety of actions for given states during training, thus filling its replay memory with a higher variety of actions, resulting in a more diverse distribution to be sampled and learned from. The action diversity produced by the entropy and stochasticity becomes more prevalent as training goes on as DDPG starts to converge towards determinism, resulting in much less diverse pools of samples.

3.5 Naive object center-point detection procedure

As object detection is not the main topic of this project, a naive solution to finding a center-point of a target object is presented. The procedure

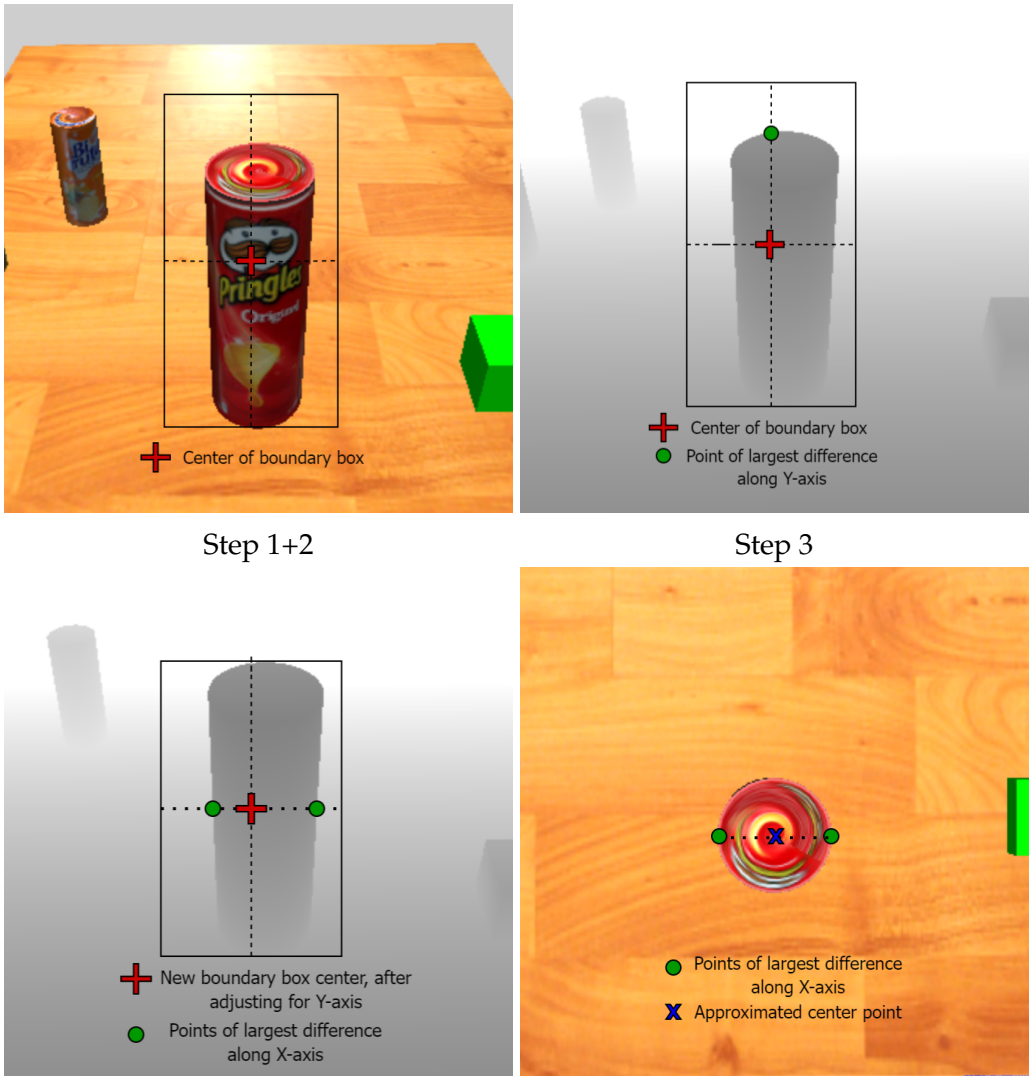


Figure 3.9: Mapping of object detection to center-point coordinates

assumes that the object is uniform and utilizes YOLO (sec. 2.6.2) object detection in addition to TIAGo's RGB-D camera. It utilized the inaccurate boundary boxes generated by YOLO to approximate the left and right edges of the object, which are then used to calculate an approximated center-point. The procedure is as follows:
Step 1: Use YOLO object detection to generate boundary boxes of the objects seen by TIAGo's camera. Multiple objects can be detected at once, so the correct one has to be chosen. YOLO's generated boundary boxes are too large and sometimes inaccurate and thus have to be shrunken down to .
Step 2: Find the center pixel coordinate of the generated boundary box.
Step 3: Use the center pixel coordinate and boundary box to scan vertically

along the y-axis of the depth image, through the center pixel coordinate, inside the boundary box. Compare neighboring depth values and store the index where the largest change happened. This is the true top of the object.
Step 4: Calculate the new center value for the Y coordinate using the true top of the object found in step 3. Scan horizontally through this Y-value, inside the boundary box. Compare neighboring depth values and store the indexes of the two places where the depth value changed the most between two neighboring measurements, these represent the left and right edges of the object.
Step 5: Use the depth camera and coordinate frame transformations available through ROS TF to calculate the two horizontal edge points in the coordinate frame of the base of the robot. The midpoint between the two calculated horizontal edge points is the approximated center-point of the object.

Algorithm 1 DDPG *as described in [1] and [7]*

Initialize batch size B, smoothing coefficient τ and discount factor γ.
Initialize critic network $Q(s, a|\theta^Q)$, and actor network $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \rightarrow \theta^Q$, $\theta^{\mu'} \rightarrow \theta^\mu$
Initialize replay memory D
for each episode **do**
  Initialize exploration noise N
  Receive initial observation s_0
  for each time step t **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + N_t$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    if time to update **then**
      Sample random batch of B transitions (s_i, a_i, r_i, s_{i+1}) from D
      Compute targets:

$$y_i = r_i + \gamma Q'(s_{t+1}, \mu'(s_{t+1}))$$

      Update critic by one step of gradient descent using:

$$\nabla_{\theta^Q} \frac{1}{B} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

      Update the actor by one step of gradient ascent using:

$$\nabla_{\theta^\mu} \frac{1}{B} \sum_i Q(s, \mu(s))$$

      Update the target networks:
      $\theta^{Q'} \rightarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
      $\theta^{\mu'} \rightarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end if
  end for
end for

Algorithm 2 SAC *as described in [1] and [7]*

Initialize batch size B, smoothing coefficient τ and discount factor γ.
Initialize critic networks $Q_1(s,a|\theta^{Q_1})$, $Q_2(s,a|\theta^{Q_2})$, and actor network $\pi(s|\theta^{\mu})$ with weights θ^{Q_1}, θ^{Q_1} and θ^{π}.
Initialize target networks Q_1' and Q_2' with weights $\theta^{Q_1'} \rightarrow \theta^{Q_1}$ and$\theta^{Q_2'} \rightarrow \theta^{Q_2}$
Initialize replay memory D
for each episode **do**
  Receive initial observation s_0
  for each time step t **do**
    Select action $a_t \sim \pi(s_t|\theta^{\pi})$
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    if time to update **then**
      Sample random batch of B transitions (s_i, a_i, r_i, s_{i+1}) from D
      Compute targets:

$$y_i = r_i + \gamma(\min_{j=1,2} Q_j'(s_{t+1}, \tilde{a}_{t+1}) - \alpha \log \pi(\tilde{a}_{t+1}|s_{t+1})),\ \tilde{a}_{t+1} \sim \pi(\cdot|s_{t+1})$$

      Update critics by one step of gradient descent using:

$$\nabla_{\theta^{Q_j}} \frac{1}{B} \sum_i (y_i - Q_j(s_i, a_i|\theta^{Q}))^2,\ for\ j = 1, 2$$

      Update the actor by one step of gradient ascent using:

$$\nabla_{\theta^{\mu}} \frac{1}{B} \sum_i (\min_{j=1,2} Q_j(s, \tilde{a}(s)) - \alpha \log \pi(\tilde{a}(s)|s))$$

      Update the target networks:

$$\theta^{Q_j'} \rightarrow \tau\theta^{Q_j} + (1-\tau)\theta^{Q_j'},\ for\ j = 1, 2$$

    end if
  end for
end for

Chapter 4

Preliminary experiments

4.1 Simulation environment

The Gazebo simulation environment is set up as shown in fig. 4.1. TIAGo is placed in front of a table where four objects are randomly initialized and TIAGo's arm is extended to its side. This is the position each episode starts in. The objects on the table are randomly shuffled between each episode, and the arm is randomly reset to be extended at either side.
At the start of each episode, TIAGo collects visual information from its depth camera to generate the point cloud that is utilized in the reward function structure designed for simulation in the Gazebo environment, as described in sec. 3.3.5. The agent also gathers the center-point position of the target object it is supposed to move the gripper to directly from the simulated environment. The agent then starts training, choosing actions defined by its policy. Each episode terminates when the empty space between the tip of the grippers is close to the target object or a threshold is met for the number of steps an agent is allowed to perform before resetting, the closeness threshold required for episode termination is tunable.

4.2 Early stage simulations

The early-stage simulations consisted of simulations in Gazebo where parameters were not properly tuned. The action space was as described in sec. 3.3.3, and did not utilize the action space changes described in sec. 3.3.4 to reduce volatile movements. The reward function was as presented in sec. 3.3.5, accompanied by the state space presented in sec. 3.3.2 for simulations in Gazebo. Actions were chosen at a rate of 2 Hz.
The initial simulated tests were done to observe and iterate over possible configurations and parameter tunings and to see if the learned policy of the agent became as intended. The runs are included to highlight the initial failures encountered when attempting to train the reinforcement

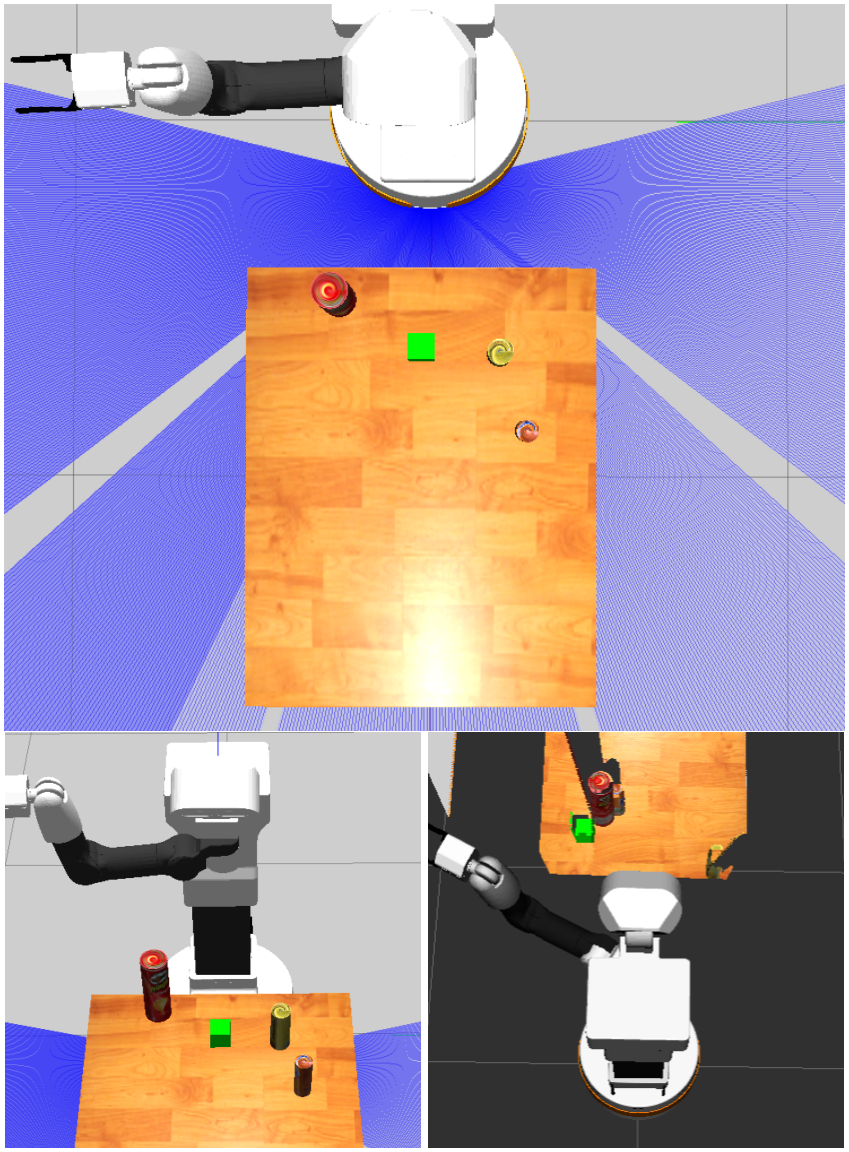


Figure 4.1: Gazebo simulation environment

learning agents to control the manipulator and why the failures led to the introduction of a second simulation environment.

4.2.1 First round

The first round of simulation was done using a SAC-based RL agent. It was done to validate if the environment was set up correctly to enable learning. It was ran until the average episodic rewards started to slightly converge, then tested to see if the end-effector of TIAGo's manipulator arm could reach a variety of object placements on the table. In this initial simulation, the threshold for closeness to the object was somewhat relaxed as the purpose of it was to validate learning and to gain insights for further

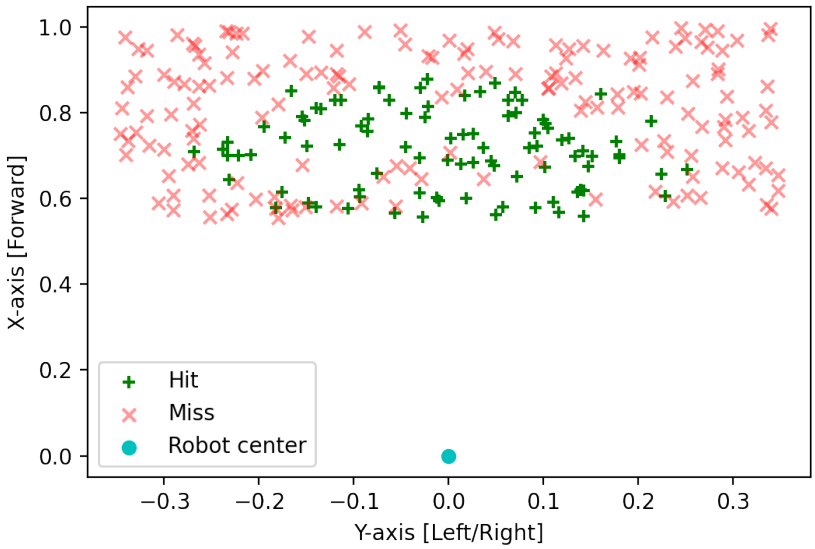


Figure 4.2: Closeness to object placement in initial simulation.

training. In fig. 4.2 it is shown at which x- and y- target object positions the arm was able to get close to. A 'hit' was defined as the end-effector being less than ten centimeters from the center-point of the target object. In table 4.1, some key insights from the initial observations can be found.

Key insights	- The agent learned to stop twisting and displacing itself by picking motions that would not lead to collisions with the table and the objects on it. - The agent learned to pick smaller movements so that it mostly stopped being penalized for planning motions that were too large for the time frame of each step. - The agent learned to somewhat follow the vector pointing to the target object, especially if the object was placed somewhere near the middle of the table.
Shortcomings	- The agent did not learn to move toward the target object if it was placed on the side of the table. - The agent did not learn to slow down or emphasize getting the grippers around the target object in the cases where it got close to it, resulting in the target object being displaced. - The accuracy of getting close to the target object was low in the global context, but higher for a clearly defined region of the table.

Table 4.1: Key points and shortcomings of initial simulation.

4.2.2 Second round

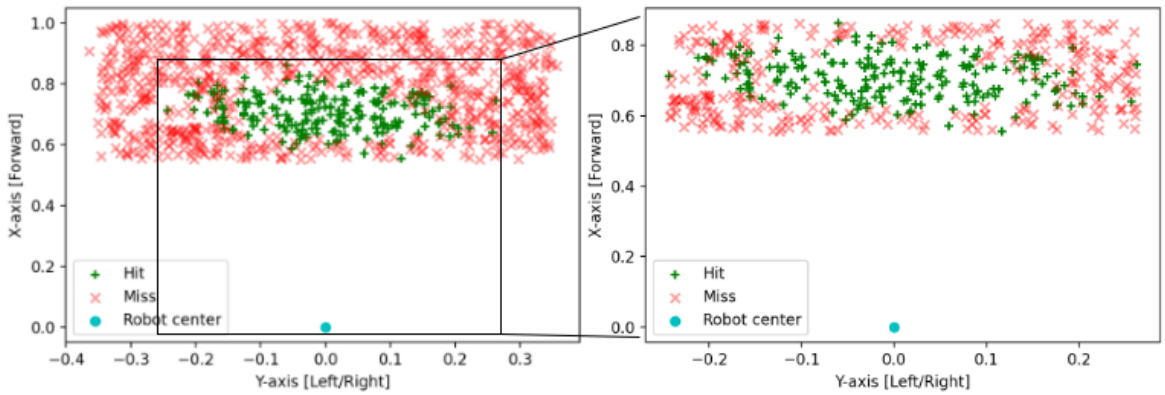


Figure 4.3: Closeness to object placements in the second round of simulation results

After observing the training of the first round of the simulations, a second round was started from scratch. Three major changes were made. Firstly, the definition of closeness was changed, a 'hit' now constituted the empty space between the end-effector's grippers being within five centimeters of the center of the object. Secondly, some hyperparameters were tuned, such as the coefficients of the reward function, to put more weight on the part that concerned reaching the target object, rather than the parts that concern collision avoidance and sudden movements. This was done because it could be observed that the agent quickly learned to move the arm in ways that negated being punished for collision or sudden movements, but was somewhat lacking in following the target object. Thirdly, the agent would be trained for a longer period of time.
In fig. 4.3, 1000 subsequent episodes are shown that were captured at the end of training. On the left side of the figure, the full range of possible object placements on the table is shown, on the right side, only the ranges where the agent was successful at least once are shown. The global hit accuracy was 20.4%, while the hit accuracy in the restricted area was 40.1%. It became quite evident from the recorded data that the agent was not able to generalize enough for the whole range of possible object placements on the table.

4.2.3 Third round

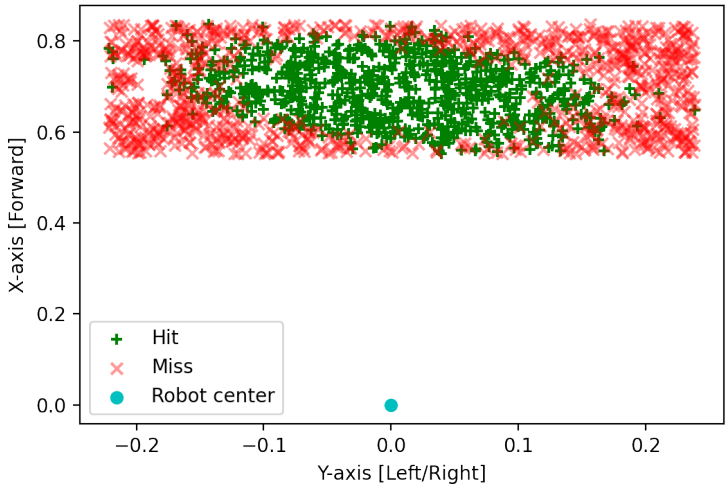


Figure 4.4: Closeness to object placements in the third round of simulation results

With the observations of the second round of simulations in mind, a third was started. Two major changes were made, both as attempts to reduce redundancy. Firstly, a change was made to the agent's state space. The first column of the $8X4$ state space matrix described in sec. 3.3.2 was removed. The change was made because each row contained both the distance from the links of the arm to either the target object or some hindrance in the environment and a vector from the same point on the arm to the same environmental object. Since the vector itself also inherently describes the distance, having the distance included separately was redundant, leaving the state space description as follows:

$$S = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \\ x_4 & y_4 & z_4 \\ x_5 & y_5 & z_5 \\ x_6 & y_6 & z_6 \\ x_7 & y_7 & z_7 \end{bmatrix}$$

Where the first row describes the vector from the end-effector to the goal object and each subsequent row describes a vector from each of the manipulator arm's coordinate frames to the closest obstacle in the point cloud. Secondly, the range of possible x- and y-coordinates for the objects on the table were reduced to fit the x- and y-ranges where the agent was

able to reach the object in the previous simulation round. This was done to reduce redundant training rounds in which the agent previously had proven to be unsuccessful.
In fig. 4.4, the results of these changes can be seen. The 'hit' accuracy in the area improved from 40.1% in the previous round, to 49.2%.

4.3 Conclusions of the early results

What became evident after observing the results and emergent behavior of the agent was that larger changes had to be made if the agent controlling the robotic arm were to be able to learn how to control the arm in a manner that would enable gripping. Simulating in Gazebo was also too slow for rapid prototyping and testing of different state-space and reward structures. This led to the implementation of the PyBullet simulation environment, described in sec. 3.2. The change in simulation environments also required the goal of the agent to be changed. Instead of focusing on teaching the agent to control the manipulator arm to move toward a target object while simultaneously avoiding other obstacles, the goal was changed to teaching the agent controlling the manipulator to move toward the target object and assume a pre-grasp position. This describes a motion in which the grippers of the manipulator's arm are placed around the target object in a manner that does not displace it and allows for further manipulation. The main reason for the change of goal was the inability of the PyBullet simulation environment to accurately simulate TIAGo's RGB-D camera.

Chapter 5

Results

In this section, the final results of the trained agents are presented. In addition to the agents being initially trained in a different simulation environment than the previous section, the action space definition is also changed in the manner described in sec. 3.3.4. The state spaces and reward functions are also changed according to the definitions presented for the PyBullet environment, presented in sections 3.3.2 and 3.3.5. The changes in state spaces and reward functions represent the goal of teaching the agent to control the manipulator to assume a pre-grasp position, while not being concerned about avoiding obstacles.
Additionally, transfer learning is attempted as the trained agents of the PyBullet environment are transferred and tested in the Gazebo environment. The best-performing agent is additionally retrained in the Gazebo environment and its performance is compared to a motion planning algorithm that is implemented in TIAGo's software packages. Lastly, the object center-point approximation procedure's (sec. 3.5) accuracy is compared to obtaining the target object's center-point directly from the simulated environment, as a center-point approximation procedure would be necessary to deploy the trained agents on a real-world setup.

5.1 Training data

Figure 5.2: Avg. reward of the agents during training

Figure 5.3: Normalized reward comparison of agents

The initial training is done in the PyBullet environment. The state space of the agents denoted 'position only' are $s = \{q, \dot{q}, p_{ee}, \bar{p}_o\}$, as presented in sec. 3.3.2. The state space of the agents denoted 'position and angular' are the same as the agents denoted 'position only' except that it also contains the angular differences between the end-effector and the object, α and θ, described in sec. 3.3.5.
The action space is as described in sec. 3.3.4, with an action space constraining coefficient of 0.1 and the reward functions are as described in sec. 3.3.5, with weighting tuned to output rewards in the range $[-50, 50]$.
Each episode is set to terminate either when the grippers assume a pre-grasp position, measured as the empty space between the end-effector grippers being within 2 cm of the center of the object, or when 40 timesteps have passed. A time step is set to be a quarter of a second. The agent thus reevaluates the environment and chooses the next action at a rate of 4 Hz. The environment is reset after each episode, placing TIAGo in its initial pose, and randomly shuffling the table object in the same x- and y-ranges as in the preliminary experiments.
The training data represented in fig. 5.2 shows how the average reward of the training episodes evolves over time. It can be observed that, as time goes on, all of the agents reach comparable performance in a similar amount of time. However, both the DDPG agents' performances eventually degenerate, and they are unable to return to performances similar to their peaks. On the other hand, both SAC agents reach their optimal policy and mostly stay there. Observing the plot for the 'SAC position only' agent, it is shown that the agent sometimes escapes its optima, but is quickly able to find its way back. In fig. 5.3, the normalized reward curves is compared, where an episodic reward of 1.0 represents the maximum achieved episodic reward any of the agents received in a single episode.

5.2 Simulated performance test

The results found in table 5.1 are gathered by running 1000 episodes of each respective trained agent with no exploration and no updates. The agents are originally trained in the PyBullet environment and the results from the Gazebo environment utilize the trained weights of the neural networks trained in the PyBullet environment. Both the SAC agents tested are using the state of the agent at the end of the training, while both the DDPG agents are using the state of the agent as it was when each of them was at their maximum avg. episode reward as seen in fig. 5.2. This choice is made because the SAC agents' performances are at a maximum at the end of their respective training runs, while the performance of both the DDPG agents' deteriorate after a certain amount of time.

Simulation environment	Agent algorithm	Episode completion(%)	Pre-grasp success(%)	Avg. time
PyBullet	SAC position only	100	86.3	12.414 steps / 3.103 s
	SAC position and angular	100	92.0	9.924 steps / 2.481 s
	DDPG position only	78.3	0.6	21.834 steps / 5.459 s
	DDPG position and angular	98.7	46.2	12.382 steps / 3.095 s
Gazebo (direct transfer from bullet)	SAC position only	78.9	2.9	10.371 steps / 2.593 s
	SAC position and angular	90.6	0.8	14.004 steps / 3.501 s
	DDPG position only	18.3	0.0	26.342 steps / 6.586 s
	DDPG position and angular	2.1	0.0	24.429 steps / 6.107 s

Table 5.1: Performance results of the fully trained agents

Simulation environment: The environment in which the test was conducted, not necessarily the environment the agent was trained in.
Agent algorithm: The algorithm and version used to train the agent.
Episode completion: The number of times the agent was able to complete an episode before the step limit. Completion is set to when the empty space between the end-effectors' grippers is within a radius of 2 cm from the perceived center of the object. The spatial coordinates of the object center are found at the beginning of the episode and remain static, thus if the object is moved by the robotic arm during the episode, the perceived center is not moved with it, meaning that the object might not be within the grippers at the termination of an episode.
Pre-grasp success: The number of times the episode terminated with an ideal result. Measured as the empty space between the end-effector's grippers being within a radius of 3 cm from the center of the object at the termination of the episode. In this measurement, the center of the object is gathered directly from the simulation environment, meaning that in this scenario the agent was able to place the grippers around the object in a manner that did not knock over or displace it, resulting in it still being between the gripper fingers.
Avg. time: The average time of completed episodes, counted as the average number of steps used in episodes the agent was able to terminate before the step limit.

5.3 Retraining results

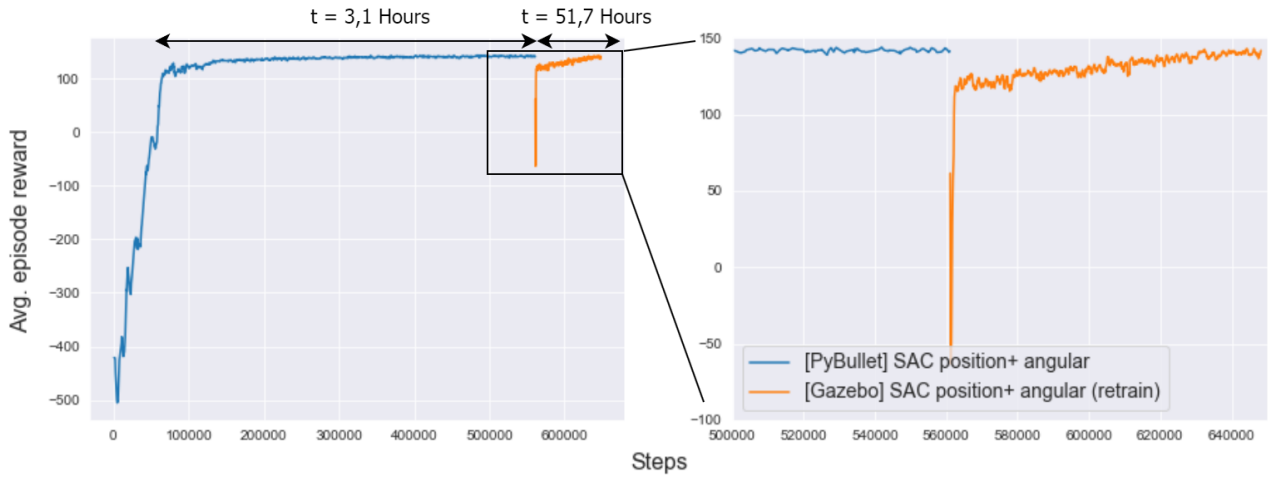


Figure 5.4: Avg. reward of the retrained agent

The best-performing agent trained in PyBullet is also retrained in the Gazebo environment. The training results are shown in fig. 5.4, and the performance test results are found in Table 5.2. Only the best-performing agent is chosen for retraining, as retraining in Gazebo takes several days. Retraining consists of loading the previously trained agent's weights into Gazebo and initiating with all parameters set to be as was when the agent's state was saved in the PyBullet environment. As shown in fig. 5.4, training the agent for $\sim$ 550 000 steps in the PyBullet environment takes 3,1 hours while retraining for $\sim$ 80 000 steps in the Gazebo environment takes 51,7 hours on the same workstation.

5.4 Example motion planning algorithm comparison

Method	**Episode completion**	**Pre-grasp success**	**Mean object displacement**	**Avg. time**
SAC agent retrained	100%	55.8%	2.54 cm	1.772 s
Example motion planning algorithm	88.2%	76.0%	0.986 cm	11.984 s

Table 5.2: Performance results of retrained agent compared to an example motion planning algorithm.

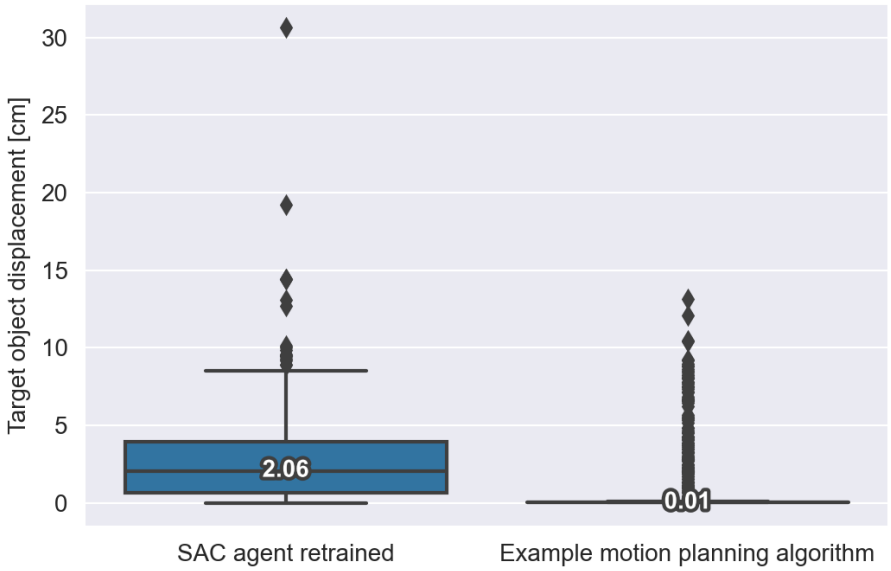


Figure 5.5: Boxplot of object displacement

The results were gathered in the same environment as the SAC agent was retrained, by placing TIAGo in front of a table and placing an object on that table. Between each reset the object is randomly shuffled within a finite range of x- and y-coordinates on the table. The motion planning algorithm (similar to what is described in sec. 2.3.4) used for comparison to the trained agent is a demo that can be found in TIAGo's software packages. The demo is described on TIAGo's wiki page[1]. In brief, the motion planning algorithm is ran by TIAGo estimating the pose of the object by a specified marker placed on top of it. The geometry of the object must be known beforehand. To make the comparison fair, the algorithm is changed to instead receive the center-point of the target object directly from the simulated environment, as the RL agent has no pose estimation component, and errors made by the example algorithm's pose estimation would be unfair to compare directly to the agent. A motion planning framework called MoveIt[8] is then called to first, estimate sample final poses of the grippers, generate trajectories that end in said final poses, and lastly choose the optimal trajectory generated. When the planning phase is over, the motion is executed. Examples of motions generated by the retrained SAC agent and the example motion planning algorithm can be found in appendix D.
In fig. 5.5 the displacement of the object from when the agent or algorithm is first called to when the agent or algorithm is in their pre-grasp positions is shown. The recorded parameters in table 5.2 are:

[1]http://wiki.ros.org/Robots/TIAGo/Tutorials/MoveIt/Pick_place

Episode completion: Measured as the empty space between the grippers being within 2 cm of the algorithm or agent's goal position. The goal position is set as the center of the object at the start of an episode.
Pre-grasp completion: Measured as the object still being within the grippers of the robot when the robot has executed its motion and is in a pre-grasp position.
Mean displacement: The mean displacement of the center of the object after the SAC agent or the motion planning algorithm is done placing the grippers in their pre-grasp positions.
Avg. time: The average amount of time used by either the SAC agent or the motion planning algorithm from when it is first called, to the grippers are placed in their respective pre-grasp positions. Note that the average time of the example motion planning algorithm is hard to measure, as calling it includes a lengthy reset routine. The time presented in the table is found by measuring the average amount of time it takes from when the algorithm is called to when it reaches the target object, which is 33.663s, and subtracting the time of the reset sequence, which is $\sim$ 21.68s, found by capturing the motion on video and visually timing the sequence by cropping the video frame-by-frame.

5.5 Object position estimation accuracy test

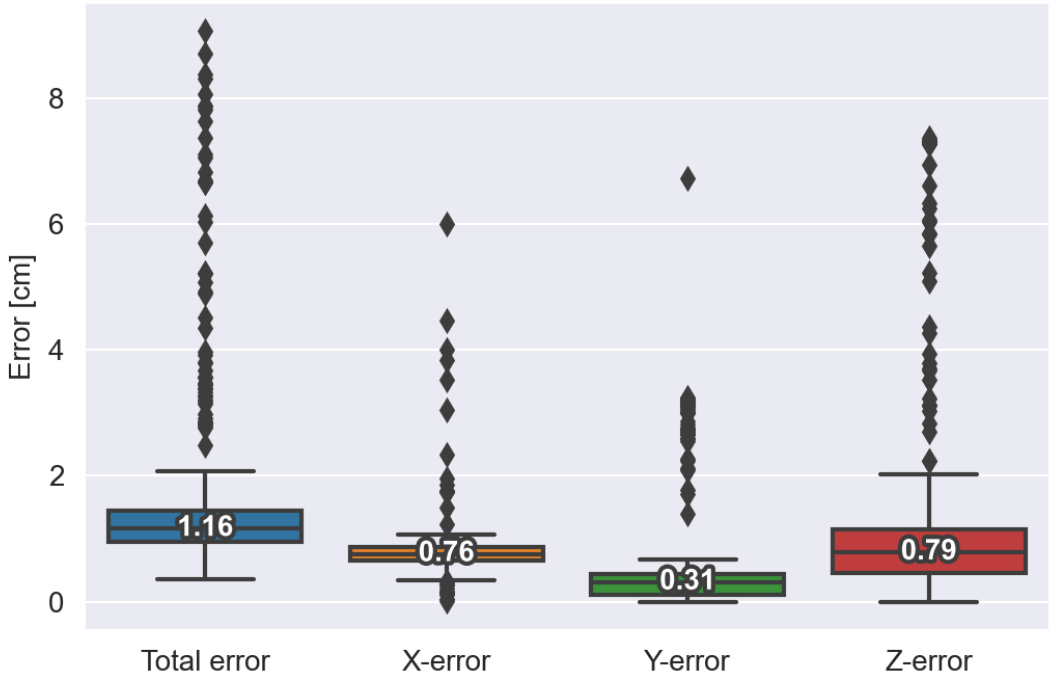


Figure 5.6: Boxplot of the object center-point approximation procedure errors

The target object center-point estimation accuracy test is conducted to observe if the center-point of the target object can be easily deduced by TIAGo's camera and a naive procedure. This is to emphasize that even though the agents are trained by gathering the center-point of the target object directly from the simualted environment, it could be possible to instead employ TIAGo's RGB-D camera to estimate the value if the reinforcement learning agents are transferred to a real-world robotic setup. The test is conducted by placing TIAGo in front of a table with the camera in the same position as the reinforcement learning agent was trained from and randomly placing the same cylindrical object on the table in the same x- and y-ranges as in the reinforcement learning agent training runs. The center of the object is approximated using the procedure described in sec. 3.5 and compared to the actual center-point of the object gathered directly from the simulation environment. In fig. 5.6, the total error is presented, along with the errors for each coordinate axis.

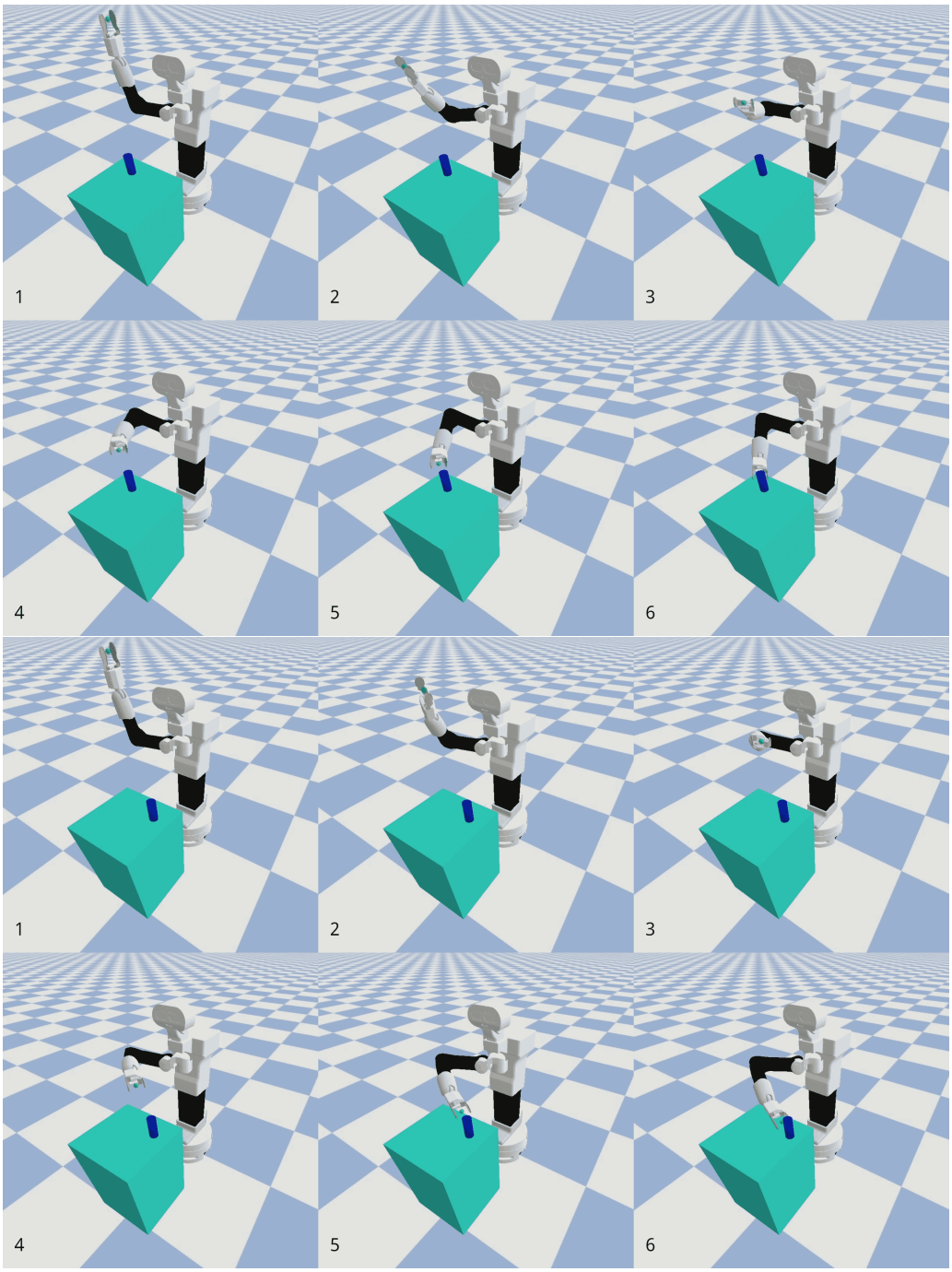


Figure 5.1: Motion sequences generated by the agent for two positions of the object

Chapter 6

Discussion and conclusion

6.1 Performance results

The performance results reported in chapter 5 and visualized in fig. 5.3 imply that the agents, when utilized at their maximum average episodic rewards, should perform similarly. However, as table 5.1 reports, it is evident that there exist significant differences in their performances. Although the SAC agent trained with only the position of the end-effector in relation to the object marginally outperforms the SAC agent trained with both the position and angle of the end-effector in terms of average episodic reward, it shows worse performance in the metrics measured in table 5.1. This is because the agent trained with both position and angle in mind is penalized if the direction of the end-effector is not optimal, leading to a marginally smaller average episodic reward for the agent. The reward curves for both DDPG agents indicate that if they are restored to the moment of their highest average episodic rewards, they should perform similarly to the SAC agents, which they do not. This observation suggests that the episodic reward return does not necessarily have a high correlation with the metrics presented in the table, even though the reward function designed to train the agents are specifically designed to encourage high performance on the specific measurements.

6.2 Simulation environments

6.2.1 Simulating in Gazebo

Due to the widespread usage of ROS in modern robotic platforms, Gazebo is often used as the default simulation environment for ROS development and is also utilized in this project. However, while Gazebo provides several advantages for robotics development, it may not be the optimal choice for reinforcement learning development. One issue is the inability to reset the robot's joints quickly, which can lead to sub-optimal reset

routines, such as having to manually move TIAGo to another place in the simulated world so the pose of the robot's arm can be manually moved to its initial position. This is done to ensure that the arm can reach its initial position without being blocked by obstacles such as the table. Routines such as this are very time-consuming and negatively impact training efficiency.

As well as having to perform convoluted reset routines to ensure that each episode starts correctly, the simulation speed of Gazebo might also impact efficient learning. As an agent requires thousands of episodes to properly learn, each taking a minute or more to complete in real-time, it would be beneficial to scale the simulation speed up to perform much faster than real-time. However, this is challenging with Gazebo since the update rate of various sensors and systems, which are pre-programmed to behave like their real-world counterparts, cannot be reliably sped up. Though the physics update rate of the simulation can be increased to achieve a higher real-time factor, it is generally considered unreliable to speed up simulations containing multiple independent control and sensory systems, such as TIAGo. Thus, it is not found safe to speed up simulations in Gazebo beyond real-time for this project, as it would require a significant degree of in-depth knowledge of the physical components, which is outside the scope.

As a result, training a reinforcement learning agent from scratch using the methods presented in this project is a time-consuming process that takes several days. This makes it challenging to prototype new ideas and tune the plethora of hyperparameters associated with training the reinforcement learning agents.

6.2.2 Simulating in PyBullet

A solution to mitigate the disadvantages of training reinforcement learning systems directly in Gazebo is to reproduce the simulation environment as closely as possible in another simulation engine. However, this approach requires simplifying the utilization of the robot's sensory systems due to the challenges of simulating the exteroceptive sensory systems without a detailed understanding of every component, which is beyond the scope of this project. Nonetheless, proprioceptive sensory systems that provide information about the internal states of the robotic manipulator, such as joint displacements and velocities, are readily available. Exteroceptive information can also be utilized if it can be ensured that the information is transferrable to the real sensory systems. For instance, the position of the center-point of the target object in the environment can be directly obtained from the simulated environment. Since the location can be estimated with the use of the robot's depth camera combined with pose estimation algorithms, the position gathered directly from the simulated envir-

onment can be used directly. However, simulating the point cloud generated by TIAGo's RGB-D camera in the PyBullet environment is a complex task, and beyond the scope of this project. Therefore, agents trained in PyBullet cannot utilize the depth information this camera provides, and thus cannot utilize the collision detection originally planned for this project as described in sec. 3.3.2, which could have been properly utilized if trained directly in Gazebo. Another advantage of running simulated experiments in the PyBullet environment is parallelism. The framework used to train the agents allows for parallel episodes to be run while training, which significantly speeds up the training process and is mostly limited by the hardware of the computer that is used.

6.2.3 Environment comparison

Simulation environment	**Trained steps**	**Time**	**Avg. steps per second**	**Most limiting factor**
PyBullet	550 000	3,1 hours	49,3 steps per second	Hardware
Gazebo	80 000	51,7 hours	0,43 steps per second	The environment

Table 6.1: Simulation environment comparison

In this project, two simulation environments are utilized, each with its own advantages and disadvantages. Fig. 5.4 shows the training process of training an agent in one environment, followed by retraining it in another. The process of training the agent in the PyBullet environment requires about 3,1 hours for approximately 550 000 steps of training while retraining in Gazebo takes about 51,7 hours for only approximately 80 000 training steps. Despite training for 550 000 steps in the first environment, it can be observed that the average episodic return begins to converge at only approximately 150 000 environmental steps and that further training only marginally improves the average return. Assuming training for only 150 000 steps, the training time would be approximately 0.8 hours. On the other hand, training the same amount of steps from scratch in Gazebo would take 96, 9 hours, assuming the average time per step found in table 6.1 is true for all phases of training. However, this might not be the case as the Gazebo environment's convoluted reset routine is a significant limiting factor. At the beginning of training, each episode is longer than in the end because the policy has not yet been trained to a point where episodes terminate before the step limit. As a result, earlier in training, the average number of steps per second is higher because the reset routine is called less often. Despite this, the time required to train an agent until the rewards start to converge shows significant benefits to utilizing

the PyBullet simulation environment instead of training the agent from scratch in Gazebo, if applicable to the problem.

6.3 Transfer leaning between the environments

Transfer learning is attempted by copying a fully trained agent from the PyBullet environment and retraining it in the Gazebo environment. Due to the long retraining time described in the previous section, only one agent is chosen for retraining. By comparing the retrained agent with its predecessor that was directly copied, but not retrained, significant performance improvements can be observed. The predecessor agent, which is not retrained, is only able to terminate 90.6 % of the episodes before the step limit and does so without displacing the object in less than 1% of the trials. On the other hand, the retrained agent is able to terminate in all tested episodes, with the object being minimally displaced and still being within the grippers in 54,5% of the tests. The retrained agent also becomes more efficient in terminating episodes and does so on average in about half the number of steps compared to before retraining. Despite the improvement in performance caused by retraining, the agent is not fully able to achieve the same accuracy it did in the PyBullet environment it was initially trained in when considering the parameters described in table 5.1.

6.4 Results comparison to example motion planning algorithm

In table 5.2, the retrained SAC agent is compared to an example motion planning algorithm from TIAGo's software packages. The results indicate that while the motion planning algorithm is slower, it achieves a higher success rate in reaching a pre-grasp position without displacing the object. It is also shown in fig. 5.5 that when the algorithm fails, the object is displaced to a lesser degree than it is when the reinforcement learning agent fails, making it more precise, even when failing. On the other hand, it is quite ineffective in comparison to the SAC agent, because the example algorithm has to first generate possible final poses of the arm, trajectories to reach them, and then find the best possible solution out of many generated samples, it has to wait before being able to execute any movement. However, if the environment is static and the robotic manipulator has to execute the same task repeatedly, the planning stage only has to be done once. A weakness of this approach is if changes happen in the environment after the planning stage is complete, as only a subtle change in the position of the object would cause the planned trajectory to no longer be up to date, and the executed motion would fail. The reinforcement learning agent has no planning stage and thus

starts moving toward the object immediately after it is called. It does on-line calculations while moving at a specified rate, in the case of the agent trained and presented here, at 4 Hz. The agent thus reevaluates the current state of TIAGo's arm in the environment and picks a new set-point configuration four times every second. The reevaluation process could potentially allow the agent to adapt to changes in the environment while they are happening, unlike the motion planning algorithm, if trained to do so.

6.5 The effect of position estimation on the fully trained agent

The results from table 5.1 were gathered using agents that obtained the position of the target object directly from the simulated environment. In fig. 5.6, the error associated with using the object center-point detection procedure described in sec. 3.5 is presented. The results show that with the naive approximation procedure, the median error is 1.16 cm from the true center of the object. The distribution of the error along the three coordinate axes is also presented. It can be observed from fig. 5.6 that the largest contributor to the total error is the error along the Z-axis. In situations where the robot's grippers are similar to those of TIAGo, the Z-axis is most resilient to errors. If errors along the x- and y-axes become significant, the grippers will displace the object. However, if the Z-error becomes large the grippers will only slide up and down along the object, but not displace it. Even so, the data presented contains several obvious outliers, and even though the median error may be acceptable for many situations, the procedure is not robust enough. Its use may result in situations where the approximated object center-point is far from the actual center.

6.6 Learned collision avoidance

The results suggest that learning collision avoidance can impact accuracy. In this project, the first reinforcement learning agents trained starts with a state space that is configured to contain information about environmental collision, using the point cloud generated by TIAGo's RGB-D camera. By providing the RL agent with a vector from every joint of the robot's arm to the closest point in the point cloud and punishing it for coming too close to said object, it quickly learns to not collide with the environment. The result is an agent that has learned to over-emphasize collision avoidance and thus restricts its movements. The results from the early simulation runs, found in sec. 4.2, suggest that the agent would rather perform small movements at the middle of the table, where the punishment for being far off from the object was statistically lowest, while not attempting to move

after the target object if it is far off from the middle. The emergent behavior makes sense considering that the table objects are uniformly distributed on the table and that if the agent has not learned to follow the object it is supposed to be targeting, the reward can be statistically maximized over a large number of episodes by targeting the middle of the table every time.
In the second iteration, the agent is trained in the PyBullet environment with no form of collision detection. Even with no collision detection implemented, the agents still learn to avoid the table in the environment. They do so, not because they learn to avoid obstacles, but because they have experienced that moving the manipulator arm in a manner that causes collisions with the table results in a lower reward feedback from the environment. The result is that the agent only has indirect knowledge of where the table is in the environment, and would not be able to avoid any additional obstacles or be able to avoid obstacles in other environments.

6.7 Configuration space as action space

In this project, the action space of all the trained reinforcement learning agents is directly linked to the configuration space C of the manipulator. In [7], a paper where the researchers train actor-critic methods on a robotic arm similar to the one attached to TIAGo, the researchers use a much less complex action space. In their project, they use an action space $A = \{\Delta x, \Delta y, \Delta z\}$, in which each value represents the change in x-, y- and z-direction the manipulator's end-effector position is to do for the next time step, and in which $\Delta x, \Delta y, \Delta z \in [-0.1, 0.1]$, as to restrict the manipulator's ability to move too far between each time step. From these action values, they use an inverse kinematics solver to calculate the configuration space parameters necessary to achieve the change in end-effector position described by their action space. They report success in teaching the manipulator to approach a virtual point in space while avoiding either static or moving obstacles, but do not attempt manipulation. While their setup seems quite effective at controlling a robotic manipulator arm and avoiding obstacles, similar setups might not work at all for manipulation tasks, because they do not consider the posture of the arm and the end-effector. For grasping and object handling, the posture of the end-effector in relation to the target object is paramount, and thus abstractions that reduce the agent's direct control of the posture of the manipulator and only consider the end-effector's position should not be made when considering training an agent for dexterous manipulation.
On the other hand, in [17] the authors explicitly state that their action space is the same as the configuration space of their robotic manipulator, a dexterous hand, and that the agent achieved the ability to learn dexterous hand manipulation with actor-critic reinforcement learning methods. While their robotic manipulator has a much smaller operating space

than what is utilized in this project and was focused on a different task, their hand's configuration space has more dimensions than TIAGo's arm has. Their results, combined with the results gathered from training the actor-critic reinforcements agents of this project, suggest that having an agent's output directly linked to the configuration space of the robotic manipulator is a good design choice for learning manipulation skills directly.

6.8 Future work

Further development of the agents presented in this project could be made. The most natural next step would be to attempt ways to further increase the accuracy of the retrained SAC agent as the accuracy drops when transferred to the Gazebo environment. If the accuracy can be successfully restored to the same level as it was in the initial simulated environment, other endeavors could be attempted. Inclusion of collision detection schemes in the reward function, such as the one described in sec. 3.3.5 could be implemented and the agent trained from scratch. This would however require work to be done on the PyBullet simulation environment, as collision detection would be dependent on exteroceptive sensory systems. The agent could also be retrained from scratch using object detection and pose estimation instead of getting the position of an object directly from the simulated environment, which would allow the agent to better adapt to the errors of the specific object detection and pose estimation method used, but this would also require the same exteroceptive sensory systems to be accurately replicated in the PyBullet simulation environment as with collision detection. It would also be interesting to attempt to train the agent to adapt to small changes in the environment while the manipulator is moving. The possibility is there due to the agent's capability of performing on-line calculations and could provide novel insights into the agent's dynamic capabilities.

6.9 Ethical considerations

The advent of more advanced and robust AI raises a number of both legal and ethical issues that are not easily resolved. Fields such as XAI (explainable AI)[3] attempt to provide a way for AI models to be understandable and not considered a 'Black-Box' where both users and sometimes even the creator may not understand how and why and AI model arrives at a given conclusion for a given input, a case that is quite common for large deep neural networks. This problem is however not as prevalent for reinforcement learning models, as they usually do not feature networks with as many layers as for instance generative networks,

and can thus offer more transparency. For the particular implementation in this project, the reinforcement learning agents can be viewed as navigating the configuration space C of the robotic manipulator's arm, and the weights of the policy networks are used to navigate the path from the initial configuration $\mathbf{q}_s$, to a final configuration $\mathbf{q}_g$ with C and $\mathbf{q}_g$ not explicitly given and the robotic manipulator's image of C being generated by the value functions of the networks.
A primary concern when developing methods that are to be used in robotic units is privacy, security, and safety. Privacy concerns the collection and handling of data gathered by the robotic unit's sensory systems. In this project, the camera is somewhat utilized in the preliminary experiments. The camera must be on to collect the point-cloud data required to determine the agent's state space. While the camera data may not be stored directly on the robot, there are still security concerns. Since the data is being recorded live, there is a possibility of hacking into the robotic unit if it is connected to the internet or other devices in which its data may be vulnerable. As a result, security protocols may need to be established if agents such as this are to be deployed on robotic units, although it is not part of this project. It is also important to consider the potential safety risks associated with deployment. Deployed reinforcement learning models may potentially damage property or cause harm if utilized in a manner in which the agents were not originally trained. It is also important to note that an uninformed user might mistakenly assume that the results presented for the agents trained in this project can be generalized to other objects or environments. Presumptions such as this might result in the agents being utilized in a manner they were not trained for and thus control the robot to perform unpredictable actions.

6.10 Conclusion

In this project, a number of different approaches have been applied to train an agent to control a manipulator arm attached to a mobile robot using actor-critic reinforcement learning methods. The agents are first trained in the Gazebo simulation environment, where the robot's sensory suite is fully implemented, to reach for a target object placed on a table while avoiding collisions with other objects on the table and the table itself. The results show that the agent fails to learn to reach the target object but learns to control the manipulator arm to avoid collisions with the environment and itself.

Later, the simulation environment is changed from Gazebo to PyBullet, and agents are trained using two slightly differing actor-critic reinforcement learning approaches to learn to control the manipulator arm to reach for and assume a pre-grasp position around a target object. One of

the trained agents is shown to be very successful at performing the task. The successful agent is transferred to and retrained in the original Gazebo environment and compared to an example motion planning algorithm.

The results from the comparison between the best-performing agent in the Gazebo environment and an example motion planning algorithm show that while the agent is more effective at reaching the pre-grasp position quickly, the motion planning algorithm is more accurate at performing the same task. In addition, the target object is shown to be less displaced when the motion planning algorithm fails than when the trained agent fails, indicating that the motion planning algorithm may be safer to use in situations where a failure could be dangerous or expensive. However, the accuracy of the best-performing agent before it is transferred to the Gazebo environment indicates that it is possible to train the agent to be more accurate than the example motion planning algorithm for the task presented.

Overall, this study shows that it is possible to train reinforcement learning agents to control TIAGo's manipulator arm to reach for objects, although the accuracy of the trained agents may be lower than that of traditional motion planning algorithms. Additionally, it highlights the trade-offs of the simulation environments employed, showing that being able to rapidly train agents might be more important than being able to utilize the full sensory suite of the robot. It enables rapid prototyping of various state space definitions, action space definitions, and reward functions, as well as making the process of tuning the hyperparameters associated with complex reinforcement learning tasks, of which there are many, more effective.

Appendix A

Packages and setup

Ubuntu Linux version	18.04.2
ROS version	Melodic
Python version(s)	2.7, 3.8
Required Python packages	(Standard ROS packages and libraries for Python), OpenAI Gym, OpenCV, Ray RLlib, Numpy, Tensorflow (Or Torch).

Table A.1: Programs used in setup

ROS and TIAGo's ROS implementation is installed on a fresh install of Linux as described on the TIAGo ROS wiki[1]. Different Python versions are installed as some of the standard ROS packages for Python are deprecated for version 3.+, while some of the packages utilized for reinforcement learning require newer Python versions. Which version is needed can be seen at the top of each Python script. Note that if using libraries that are incompatible with the same version of Python, it is possible to make two separate scripts and have them talk to each other through the ROS topics and messaging system. This is widely used in the scripts.

A.1 Python setup for RLlib

After installing the ROS TIAGo packages it is necessary to install a couple of python packages to make the simulations work properly, some of the packages are highly version-specific and will throw errors if not installed correctly. Start by installing python 3.8 as described in the article linked here [2].

[1] https://wiki.ros.org/Robots/TIAGo/Tutorials/
[2] https://linuxize.com/post/how-to-install-python-3-8-on-ubuntu-18-04/

This will install Python 3.8 but not overwrite the Python 2.7 version already installed, as having the version that comes with the ROS packages as a baseline is necessary for TIAGO's software packages to function properly.
Installation of the next packages needs to be done for Python 3.8 and not the version that came with ROS. Packages such as numpy, ray RLlib, TensorFlow, and TensorFlow probability need to be installed. They can be installed using the following commands:

```
$ python3.8 -m pip install tensorflow
$ python3.8 -m pip install tensorflow-probability
$ python3.8 -m pip install numpy
$ python3.8 -m pip install gym
$ python3.8 -m pip install -U "ray[default]==2.1.0"
$ python3.8 -m pip install -U "ray[rllib]==2.1.0"
```

The last part is to import the ROS package created for this project from the repository containing the code utilized for his project[3]. It is also necessary to read the README file included with the code repository, as it contains important information on some structural changes that need to be made in order for the scripts to work. Rebuild the workspace by going to the root folder and run:

```
$ catkin build
```

The repository also contains a YOLO ROS object detection implementation that works well with TIAGo, alternatively, YOLO ROS can be downloaded from the YOLO ROS repository[4], but the current main branch is not fully compatible with ROS melodic and Ubuntu 18.04.

A.2 Tutorials

A.2.1 How to run the trained agents

Note: *Due to the size of and number of files, only the agents that are fully trained and tested in the results chapter (chap. 5) are included in the repository.*
Open at least three separate terminal windows and go to the Catkin Workspace where ROS and TIAGo's components are installed. Source the setup file in all the windows.

```
$ cd {Tiago ROS catkin workspace}
$ source ./devel/setup.bash
```

In the first window, launch ROS and the simulation environment through Gazebo:

[3]https://github.uio.no/markustr/markustr-master-project-code
[4]https://github.com/leggedrobotics/darknet_ros

```
$ roslaunch tiago_rl env_simple.launch
```

Wait for the simulation environment to launch, and for TIAGo to tuck its arm.
In the second window, run:

```
$ rosrun tiago_rl init_pos.py
```

This will cause TIAGo to point its head toward the table so that it can observe it, while also moving its arm to the initial position. Wait for this command to finish, then in the same window run:

```
$ rosrun tiago_rl EVAL_env_handler_{arg}.py
```

In which {*arg*} is either 'pos_only' or 'pos_and_ang', depending on the state space structure designed for the agent that is going to be utilized. This will run a script that is used to communicate between the RL environment and Gazebo. It serves the state space by making the necessary coordinate frame transformations by communicating with the /TF topic and receiving the necessary kinematic chains.

In the last terminal window, run:

```
$ rosrun tiago_rl EVAL_{arg}.py
```

In which {*arg*} can be 'DDPG_pos_and_ang',' DDPG_pos_only', 'SAC_pos_and_ang', 'SAC_pos_only' or 'SAC_retrained', depending on which agent is going to be utilized.

A.2.2 How to visualize in rviz

There are multiple ways to visualize in ROS. This tutorial is based on rviz. Open two new terminal windows, go to your TIAGo install directory and source the environment in both.

```
$ cd {Tiago ROS catkin workspace}
$ source ./devel/setup.bash
```

In the first terminal window, run.

```
$ rosrun rviz rviz
```

Opening rviz this way will open it in a blank state, this will throw an error because no global map is selected. To fix this, go to global options and change the fixed frame from map to 'base_footprint', see fig. A.1.
After this is done we are free to add visualizations to rviz, start by adding the robot. This can be done by pressing the 'add' button and choosing 'RobotModel' in the list that pops up. A good idea might be to change the 'alpha' option of the robot from 1 to 0.5 so that it becomes somewhat transparent and it is easier to see the visual aids later. The point-cloud image generated by TIAGo's RGB-D camera can also be added, this can

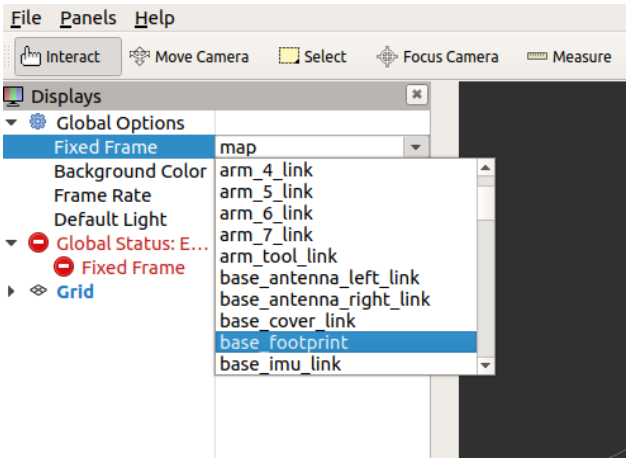


Figure A.1: Rviz interface

be done by pressing 'add', choosing to add by topics, and adding the topic '/xtion/depth_registered/points/Pointcloud2'.
Now, to add more visual aids go to the second terminal window and run: *Note that the following script requires the agent to already be running as it depends on being served the state space of the agent.*

```
$ rosrun tiago_rl frame_viz.py
```

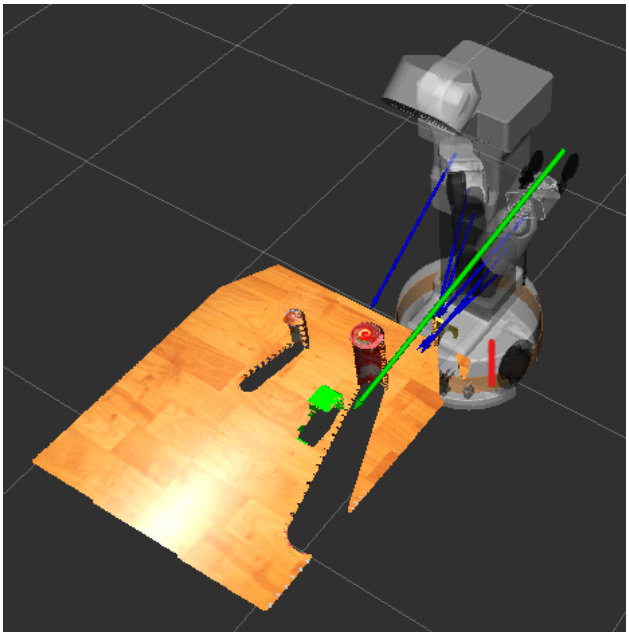


Figure A.2: Model of TIAGo in Rviz

Now go back to the rviz window and press 'add' again, choose to add by topic. This time a new option has popped up called '/env_viz_markers/Marker'. Choose to add this option. Adding this

causes rviz to show the vector from the end-effector to the target object, as well as show the coordinate frames of the end-effect, the object, and the preferred directions of the end-effector, described in sec. 3.3.5.

Appendix B

Brief forward kinematics example

B.1 The denavit-hartenberg conventions

The transformation between two coordinate frames in a robotic manipulator can be expressed as a combination of only four variables if the coordinate frames are made following these rules:

DH1: The axis of x_{i+1} is perpendicular to the axis of z_i.
DH2: The axis of x_{i+1} intersects the axis of z_i.

By following these two rules it is possible to represent the homogeneous transform between the two coordinate frames by four variables:

Link length	Link twst	Link offset	Joint angle
a_i	α_i	d_i	θ_i

Table B.1: DH parameters

Using the four variables, the homogenous transformation matrix can be represented as:

$$A = Rot_{z,\theta_i} Trans_{z,d_i} Trans_{x,a_i} Rot_{x,\alpha_i}$$

$$= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} & 0 & 0 \\ s_{\theta_i} & c_{\theta_i} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha_i} & -s_{\alpha_i} & 0 \\ 0 & s_{\alpha_i} & c_{\alpha_i} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i} c_{\alpha_i} & s_{\theta_i} s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i} c_{\alpha_i} & c_{\theta_i} s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Typically, the approach for constructing coordinate frame trees involves limiting only one of the four DH parameters to be variable between

each coordinate frame. This design allows each frame to represent either a pure rotation or a pure translation, making it necessary to utilize multiple interconnected frames to represent more complex movement. For instance, a spherical wrist that has three degrees of freedom can be modeled as three overlapping coordinate frames, each representing a single degree of freedom.

B.2 Deriving forward kinematics

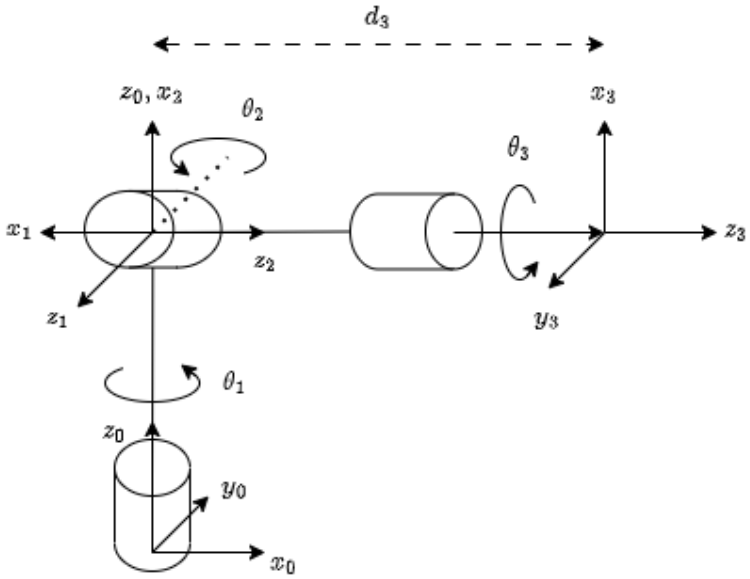


Figure B.1: Spherical wrist frame assignment. *Adapted from [40].*

Link	a_i	α_i	d_i	θ_i
1	0	$-90°$	0	θ_1^*
2	0	$90°$	0	θ_2^*
3	0	0	d_3	θ_1^*

* = variable

Table B.2: DH parameters for spherical wrist

Following the DH conventions, a set of coordinate frames has been established, making it possible to track the end-effector's position in space. By utilizing the configuration in fig. B.1 and its DH table (table B.2), it is possible to determine the homogenous transformation matrices between

each frame as:

$$A_1 = \begin{bmatrix} c_{\theta_1} & 0 & -s_{\theta_1} & 0 \\ s_{\theta_1} & 0 & c_{\theta_1} & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; A_2 = \begin{bmatrix} c_{\theta_2} & 0 & s_{\theta_2} & 0 \\ s_{\theta_2} & 0 & -c_{\theta_2} & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_3 = \begin{bmatrix} c_{\theta_3} & -s_{\theta_3} & 0 & 0 \\ s_{\theta_3} & c_{\theta_3} & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Through the use of the frame-to-frame transformations, it is possible to determine the forward kinematic transformation from the end-effector to the base by multiplying each homogenous transformation matrix A_n sequentially.

$$T_3^0 = A_1 A_2 A_3$$

$$= \begin{bmatrix} c_{\theta_1}c_{\theta_2}c_{\theta_3} - s_{\theta_1}s_{\theta_3} & -c_{\theta_1}c_{\theta_2}s_{\theta_3} - s_{\theta_1}c_{\theta_3} & c_{\theta_1}s_{\theta_2} & c_{\theta_1}s_{\theta_2}d_3 \\ s_{\theta_1}c_{\theta_2}c_{\theta_3} + c_{\theta_1}s_{\theta_3} & -s_{\theta_1}c_{\theta_2}s_{\theta_3} + c_{\theta_1}c_{\theta_3} & s_{\theta_1}s_{\theta_2} & s_{\theta_1}s_{\theta_2}d_3 \\ -s_{\theta_2}c_{\theta_3} & s_{\theta_2}s_{\theta_3} & c_{\theta_2} & c_{\theta_2}d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Additionally, this method provides a way for transforming points that are referenced in a particular frame to another. This is particularly useful in scenarios where a camera is mounted on the robot and it is necessary to express the features captured by the camera in the global context of the environment or the robot, rather than only in relation to the camera itself.

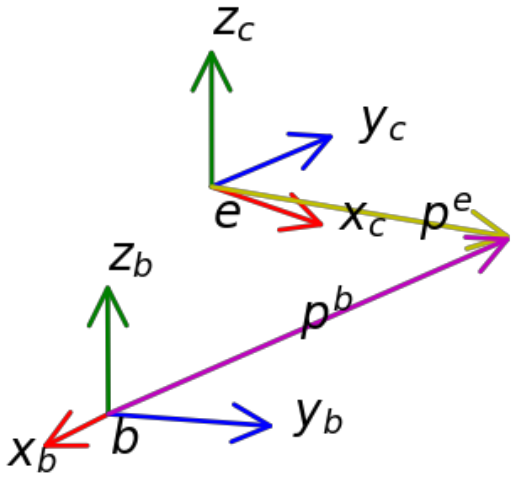


Figure B.2: Point coordinate transform

Suppose a point p^e is provided in reference to the end-effector's coordinate frame. It is possible to convert it to the base coordinate frame

by first padding it to make its shape multipliable with the transformation matrix. Next, the padded point is post-multiplied with the transformation matrix describing the homogenous transformation from the end-effector's coordinate frame to the base. Lastly, the padding is removed from the result of the multiplication, resulting in a point p^b, describing the same point as p^e, but expressed in the base coordinate frame.

$$p^e = \begin{bmatrix} x^e \\ y^e \\ z^e \end{bmatrix}, \; P^e = \begin{bmatrix} p^e \\ 1 \end{bmatrix} = \begin{bmatrix} x^e \\ y^e \\ z^e \\ 1 \end{bmatrix}$$

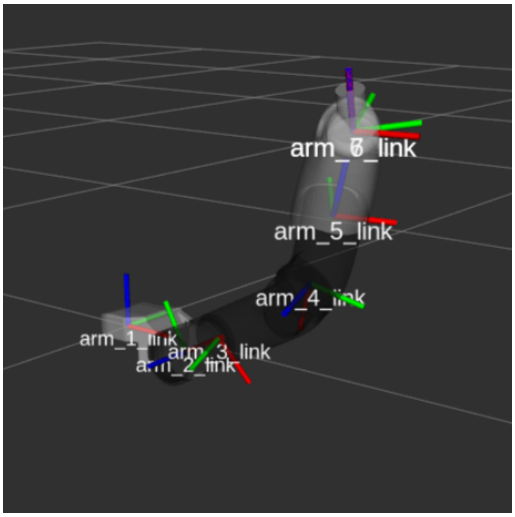
$$P^b = T_e^b P^e, \; P^b = \begin{bmatrix} p^b \\ 1 \end{bmatrix}$$

Appendix C

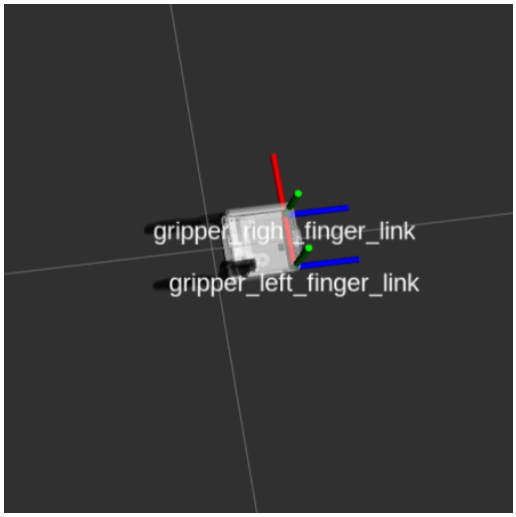
TIAGo constraints

Body part	**Link name, (P)rismatic or (R)evolute**	**Minimum displacement**	**Maximum displacement**
Head	head_1_joint(R)	-1.24 rad	1.24 rad
	head_2_joint(R)	-0.98 rad	0.72 rad
Torso	torso_lift_joint(P)	0.0 m	0.35 m
Arm	arm_1_joint (R)	0.07 rad	2.68 rad
	arm_2_joint(R)	-1.50 rad	1.02 rad
	arm_3_joint(R)	-3.46 rad	1.50 rad
	arm_4_joint(R)	-0.32 rad	2.29 rad
	arm_5_joint(R)	-2.07 rad	2.07 rad
	arm_6_joint(R)	-1.39 rad	1.39 rad
	arm_7_joint(R)	-2.07 rad	2.07 rad
Gripper	left_finger_joint(P)	0.0 m	0.04 m
	right_finger_joint(P)	0.0 m	0.04 m

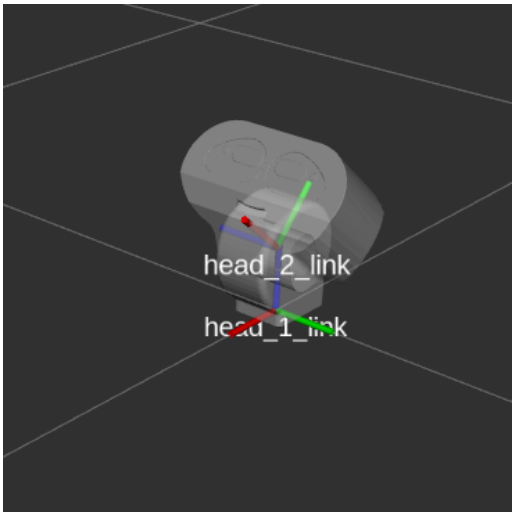
Table C.1: TIAGo's controllable joints and their constraints



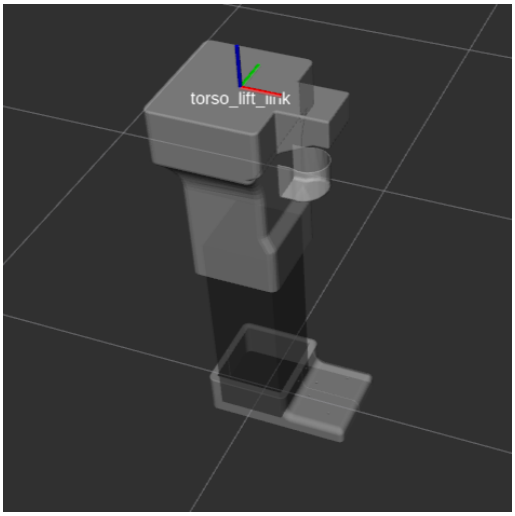
The arm, link 6 and 7 overlaps



The gripper



The head



The prismatic torso

Figure C.1: Body parts of TIAGo with frames

Appendix D

Example motions

Some executed motion sequences of the retrained SAC agent and the example motion planning algorithm discussed in sec. 5.4 are included. Each included SAC agent-generated motion consists of six frames, and each motion executed by the example motion planning algorithm consists of 12 frames, this is to emphasize that the motions executed by the example motion planning algorithm are slower than those generated by the SAC agent.

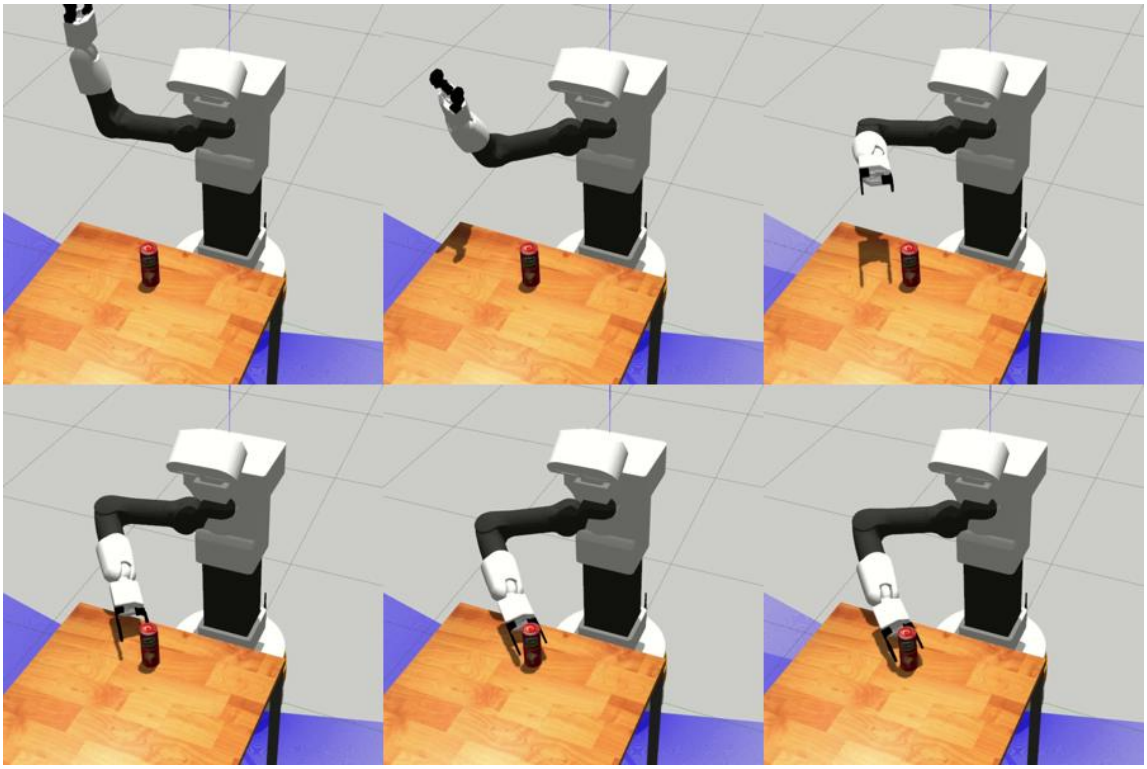


Figure D.1: Example motion executed by the retrained SAC agent

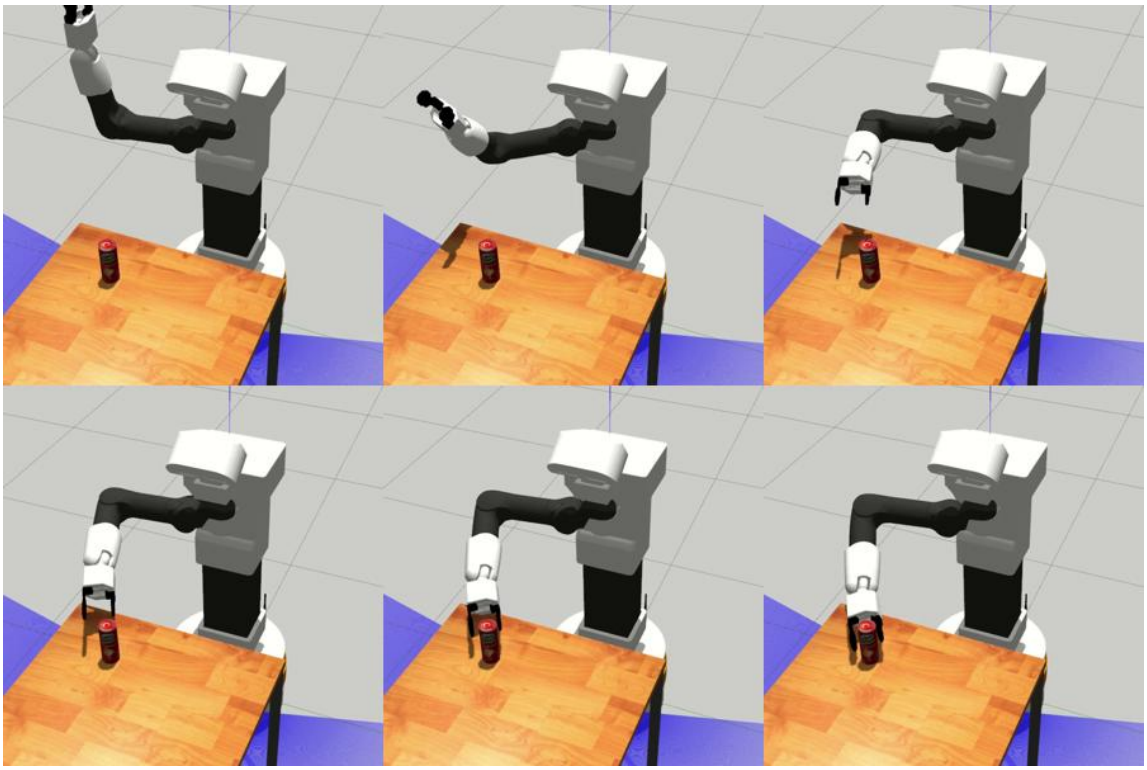


Figure D.2: Example motion executed by the retrained SAC agent

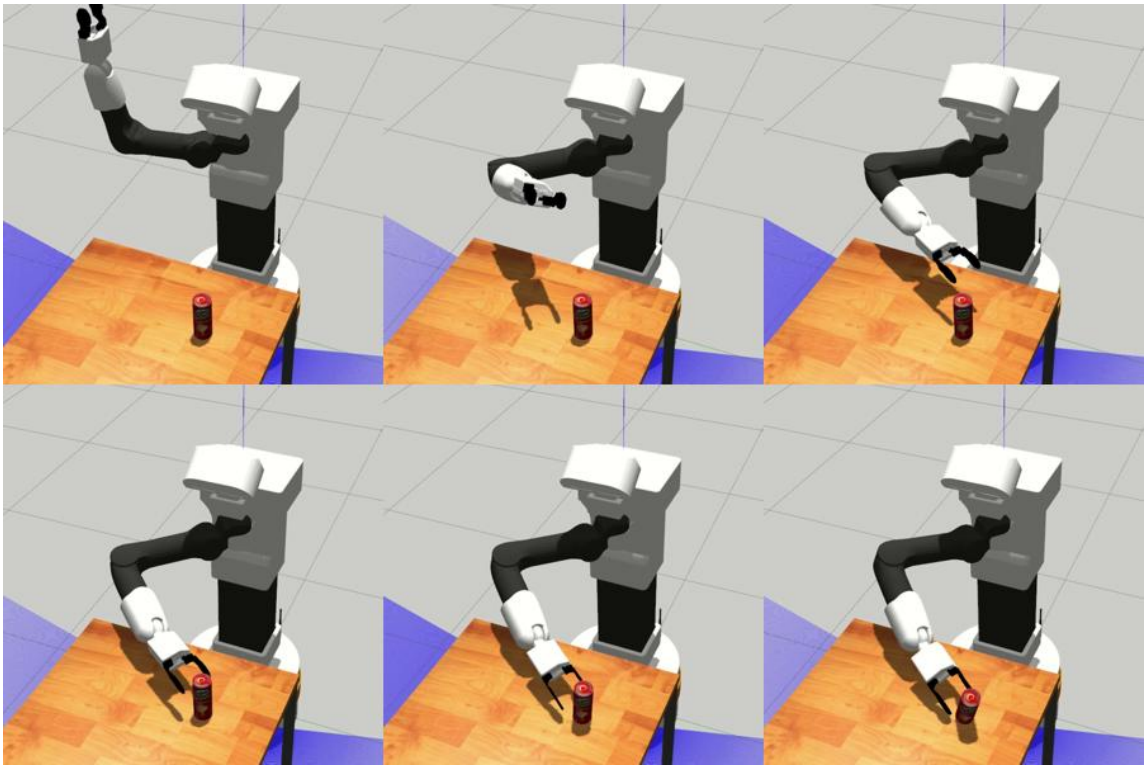


Figure D.3: Example motion executed by the retrained SAC agent

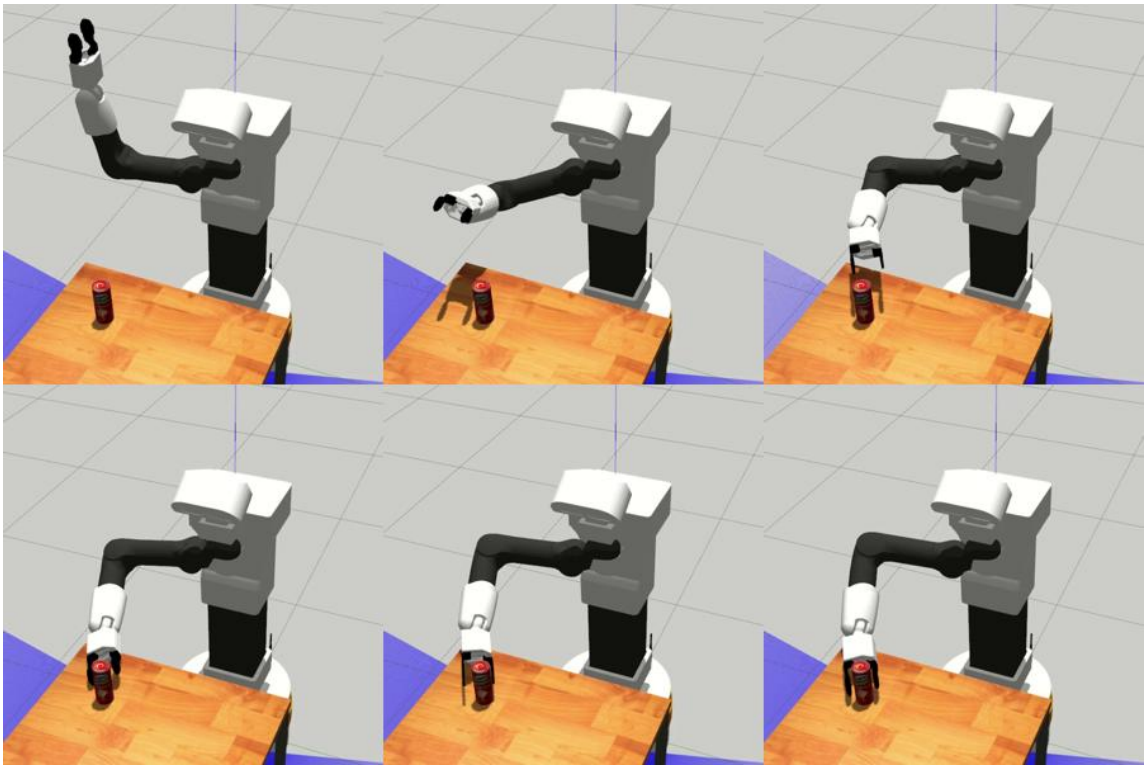


Figure D.4: Example motion executed by the retrained SAC agent

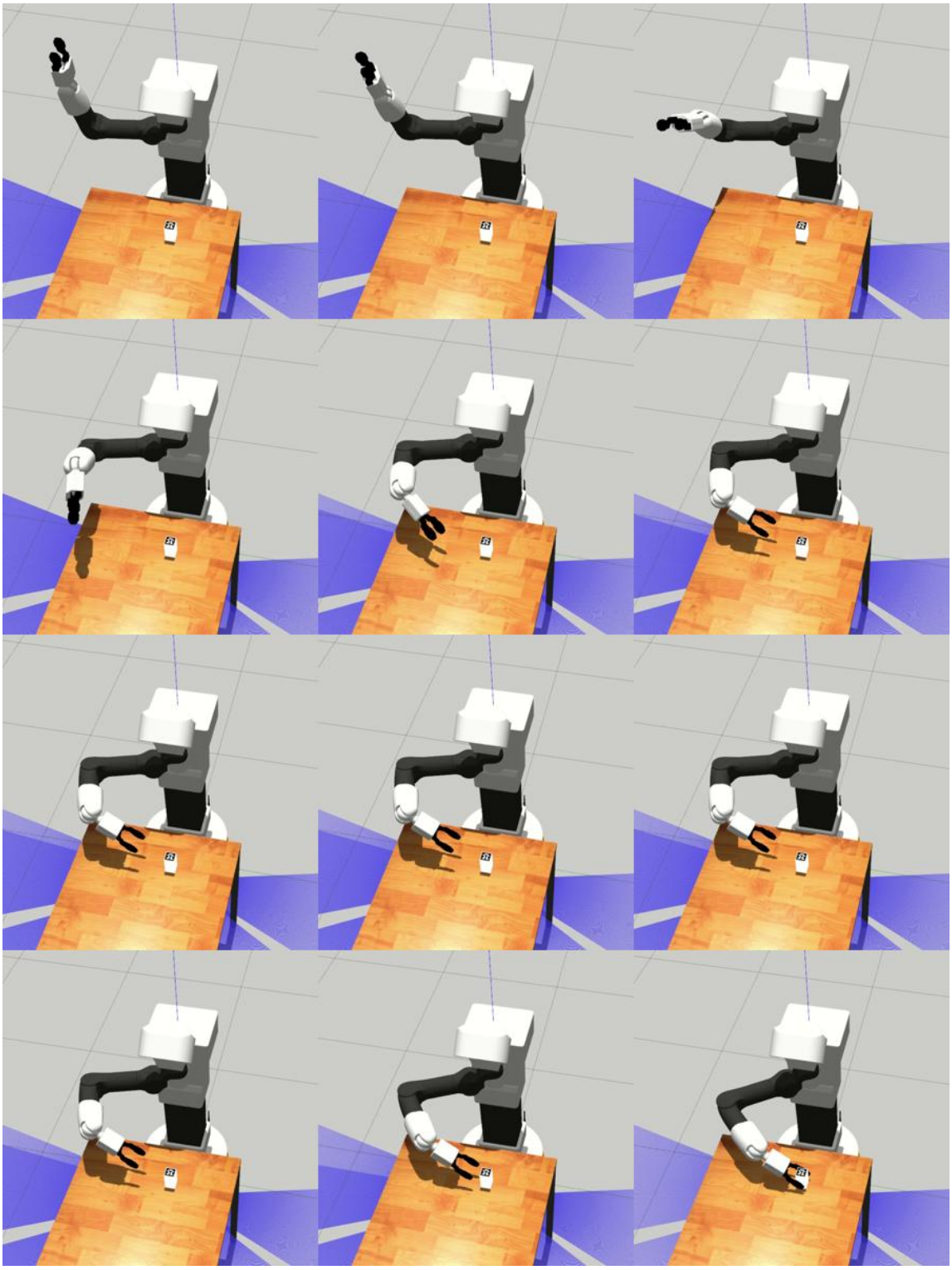


Figure D.5: Example motion executed by the example motion planning algorithm

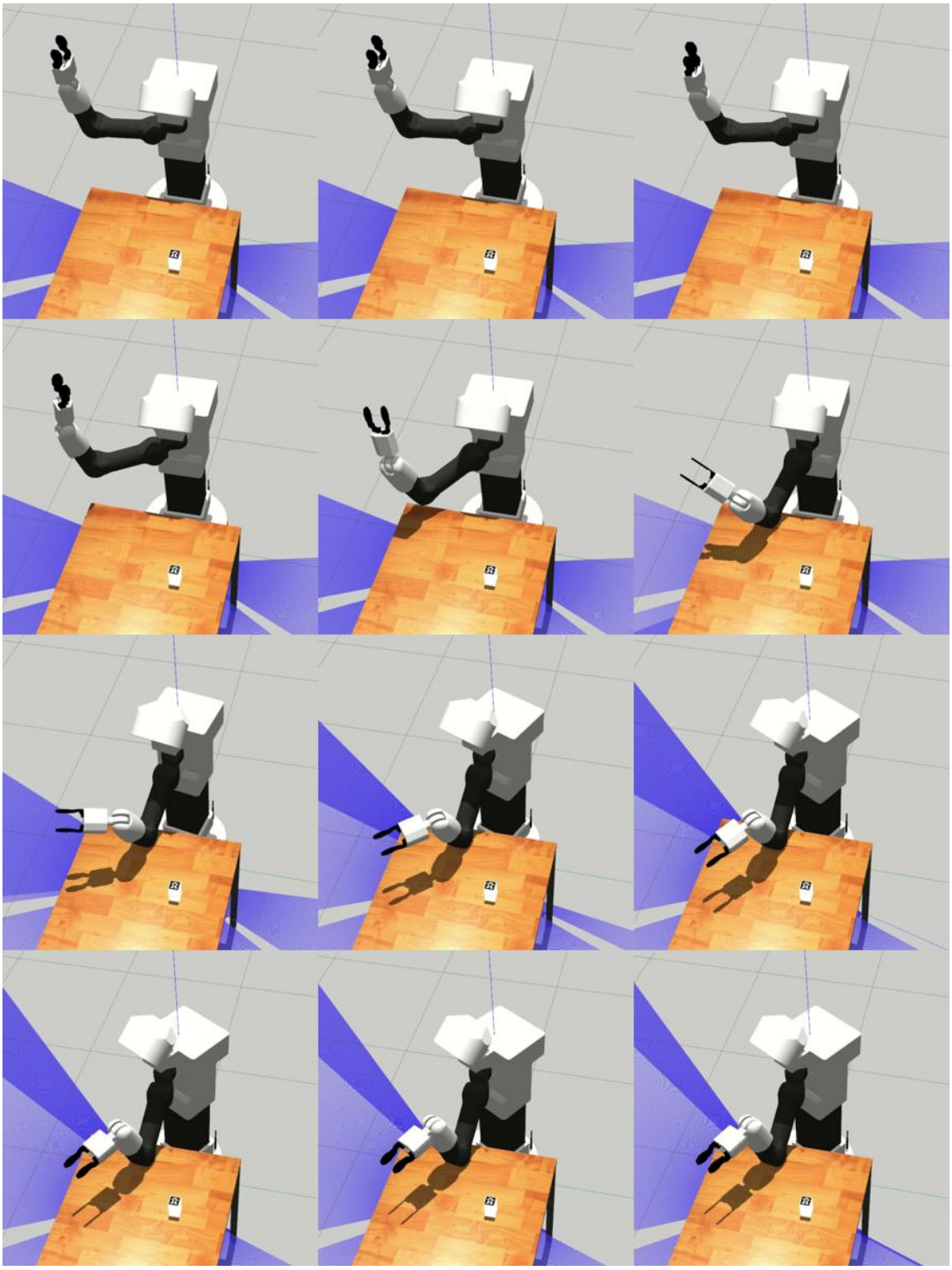


Figure D.6: Example motion executed by the example motion planning algorithm

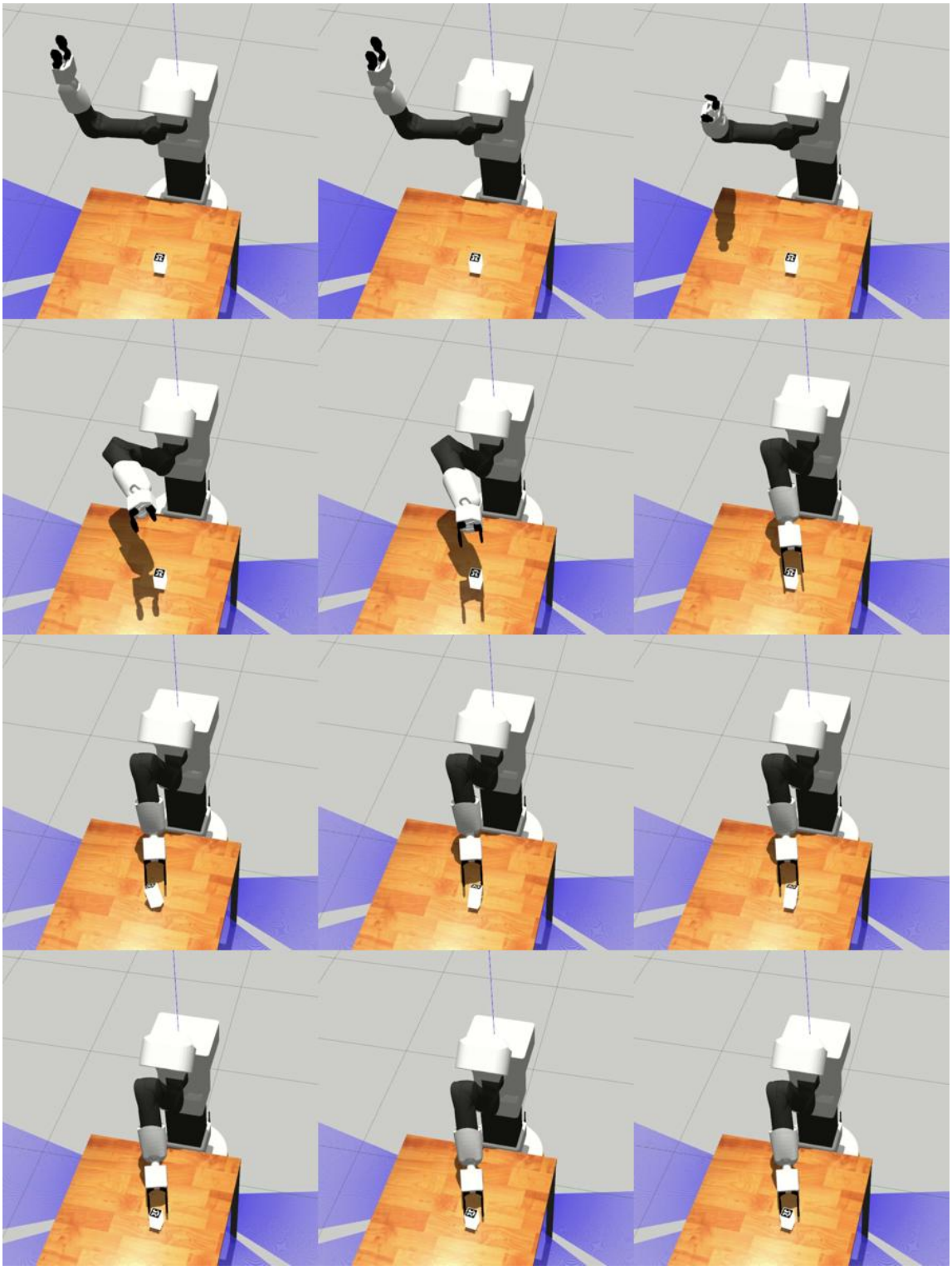


Figure D.7: Example motion executed by the example motion planning algorithm

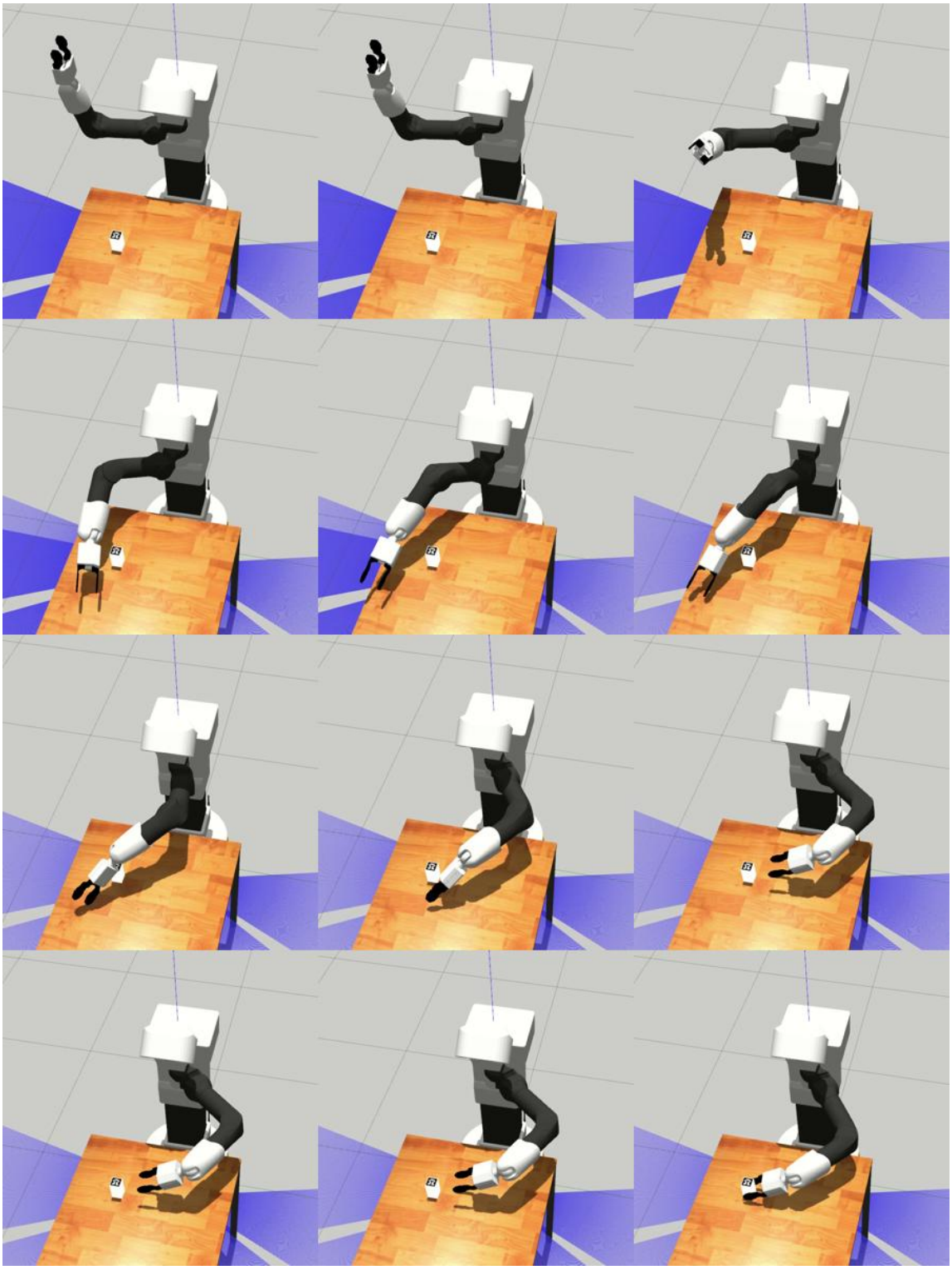


Figure D.8: Example motion executed by the example motion planning algorithm

Bibliography

[1] Joshua Achiam. 'Spinning Up in Deep Reinforcement Learning'. In: *spinningup.openai.com* (2018). URL: spinningup.openai.com.

[2] Evan Ackerman. 'Care-O-bot 4 Is the Robot Servant We All Want but Probably Can't Afford'. In: *Spectrum IEEEE* 29 (2015). URL: https://spectrum.ieee.org/care-o-bot-4-mobile-manipulator.

[3] Plamen P. Angelov et al. 'Explainable artificial intelligence: an analytical review'. In: *WIREs Data Mining and Knowledge Discovery* 11.5 (2021), e1424. DOI: https://doi.org/10.1002/widm.1424. eprint: https://wires.onlinelibrary.wiley.com/doi/pdf/10.1002/widm.1424. URL: https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1424.

[4] Kai Arulkumaran et al. 'Deep Reinforcement Learning: A Brief Survey'. In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38. DOI: 10.1109/MSP.2017.2743240.

[5] E.N. Barron and H. Ishii. 'The Bellman equation for minimizing the maximum cost'. In: *Nonlinear Analysis: Theory, Methods Applications* 13.9 (1989), pp. 1067–1090. ISSN: 0362-546X. DOI: https://doi.org/10.1016/0362-546X(89)90096-5. URL: https://www.sciencedirect.com/science/article/pii/0362546X89900965.

[6] Marko Bjelonic. *YOLO ROS: Real-Time Object Detection for ROS*. 2016–2018. URL: https://github.com/leggedrobotics/darknet_ros.

[7] Lienhung Chen et al. 'Deep Reinforcement Learning Based Trajectory Planning Under Uncertain Constraints'. In: *Frontiers in Neurorobotics* 16 (2022). ISSN: 1662-5218. DOI: 10.3389/fnbot.2022.883562. URL: https://www.frontiersin.org/articles/10.3389/fnbot.2022.883562.

[8] David Coleman et al. *Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study*. 2014. arXiv: 1404.3785 [cs.RO].

[9] Erwin Coumans and Yunfei Bai. *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. 2016–2022. URL: http://pybullet.org.

[10] Felix Endres et al. '3-D Mapping With an RGB-D Camera'. In: *IEEE Transactions on Robotics* 30.1 (2014), pp. 177–187. DOI: 10.1109/TRO.2013.2279412.

[11] David Fischinger et al. 'Hobbit, a care robot supporting independent living at home: First prototype and lessons learned'. In: *Robotics and Autonomous Systems* 75 (2016). Assistance and Service Robotics in a Human Environment, pp. 60–78. ISSN: 0921-8890. DOI: https://doi.org/10.1016/j.robot.2014.09.029. URL: https://www.sciencedirect.com/science/article/pii/S0921889014002140.

[12] Vincent François-Lavet et al. 'An Introduction to Deep Reinforcement Learning'. In: *Foundations and Trends ® in Machine Learning* 11.3-4 (2018), pp. 219–354. ISSN: 1935-8237. DOI: 10.1561/2200000071. URL: http://dx.doi.org/10.1561/2200000071.

[13] Scott Fujimoto, Herke van Hoof and David Meger. 'Addressing Function Approximation Error in Actor-Critic Methods'. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 1587–1596. URL: https://proceedings.mlr.press/v80/fujimoto18a.html.

[14] Moritz Goeldner, Cornelius Herstatt and Frank Tietze. 'The emergence of care robotics — A patent and publication analysis'. In: *Technological Forecasting and Social Change* 92 (2015), pp. 115–131. ISSN: 0040-1625. DOI: https://doi.org/10.1016/j.techfore.2014.09.005. URL: https://www.sciencedirect.com/science/article/pii/S0040162514002753.

[15] Ivo Grondman et al. 'A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients'. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595.

[16] Tuomas Haarnoja et al. *Learning to Walk via Deep Reinforcement Learning*. 2019. arXiv: 1812.11103 [cs.LG].

[17] Tuomas Haarnoja et al. *Soft Actor-Critic Algorithms and Applications*. 2019. arXiv: 1812.05905 [cs.LG].

[18] Tuomas Haarnoja et al. 'Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor'. In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 1861–1870. URL: https://proceedings.mlr.press/v80/haarnoja18b.html.

[19] Helsedirektoratet. 'Estimert mangel på helsepersonell - en grafisk framstilling av resultatene fra NAVs bedriftsundersøkelser [nettdokument]'. In: *Oslo: Helsedirektoratet* (2021). URL: https://www.helsedirektoratet.no/rapporter/estimert-mangel-pa-helsepersonell.

[20] Jennifer Hicks. *Hector: Robotic Assistance for the Elderly*. Forbes. Aug. 2012. URL: https://www.forbes.com/sites/jenniferhicks/2012/08/13/hector-robotic-assistance-for-the-elderly/?sh=d50664724437.

[21] Qingyan Huang. 'Model-Based or Model-Free, a Review of Approaches in Reinforcement Learning'. In: *2020 International Conference on Computing and Data Science (CDS)*. 2020, pp. 219–221. DOI: 10.1109/CDS49703.2020.00051.

[22] Julian Ibarz et al. 'How to train your robot with deep reinforcement learning: lessons we have learned'. In: *The International Journal of Robotics Research* 40.4-5 (2021), pp. 698–721. DOI: 10.1177/0278364920987859. eprint: https://doi.org/10.1177/0278364920987859. URL: https://doi.org/10.1177/0278364920987859.

[23] Nick Jakobi, Phil Husbands and Inman Harvey. 'Noise and the reality gap: The use of simulation in evolutionary robotics'. In: *Advances in Artificial Life*. Ed. by Federico Morán et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 704–720. ISBN: 978-3-540-49286-3.

[24] Sertac Karaman and Emilio Frazzoli. *Sampling-based Algorithms for Optimal Motion Planning*. 2011. arXiv: 1105.1186 [cs.RO].

[25] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[26] Nathan Koenig and Andrew Howard. 'Design and use paradigms for Gazebo, an open-source multi-robot simulator'. In: *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)* 3 (2004), pp. 2149–2154. DOI: 10.1109/IROS.2004.1389727.

[27] Päivi Lavander, Merja Meriläinen and Leena Turkki. 'Working time use and division of labour among nurses and health-care workers in hospitals – a systematic review'. In: *Journal of Nursing Management* 24.8 (2016), pp. 1027–1040. DOI: https://doi.org/10.1111/jonm.12423. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/jonm.12423. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/jonm.12423.

[28] Sergey Levine et al. 'Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems'. In: *CoRR* abs/2005.01643 (2020). arXiv: 2005.01643. URL: https://arxiv.org/abs/2005.01643.

[29] Shiyu Liang and R. Srikant. 'Why Deep Neural Networks?' In: *CoRR* abs/1610.04161 (2016). arXiv: 1610.04161. URL: http://arxiv.org/abs/1610.04161.

[30] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. DOI: 10.48550/ARXIV.1509.02971. URL: https://arxiv.org/abs/1509.02971.

[31] A. Rupam Mahmood et al. 'Benchmarking Reinforcement Learning Algorithms on Real-World Robots'. In: *Proceedings of The 2nd Conference on Robot Learning*. Ed. by Aude Billard et al. Vol. 87. Proceedings of Machine Learning Research. PMLR, 29–31 Oct 2018, pp. 561–591. URL: https://proceedings.mlr.press/v87/mahmood18a.html.

[32] Martijn van Otterlo and Marco Wiering. 'Reinforcement Learning and Markov Decision Processes'. In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. URL: https://doi.org/10.1007/978-3-642-27645-3_1.

[33] Cristiano Premebida, Rares Ambrus and Zoltan-Csaba Marton. 'Intelligent Robotic Perception Systems'. In: *Applications of Mobile Robots*. Ed. by Efren Gorrostieta Hurtado. Rijeka: IntechOpen, 2019. Chap. 6. DOI: 10.5772/intechopen.79742. URL: https://doi.org/10.5772/intechopen.79742.

[34] Martin L. Puterman. 'Chapter 8 Markov decision processes'. In: *Stochastic Models*. Vol. 2. Handbooks in Operations Research and Management Science. Elsevier, 1990, pp. 331–434. DOI: https://doi.org/10.1016/S0927-0507(05)80172-0. URL: https://www.sciencedirect.com/science/article/pii/S0927050705801720.

[35] Wang Qiang and Zhan Zhongli. 'Reinforcement learning model, algorithms and its application'. In: *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC)*. 2011, pp. 1143–1146. DOI: 10.1109/MEC.2011.6025669.

[36] Morgan Quigley et al. 'ROS: an open-source Robot Operating System'. In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[37] Joseph Redmon et al. 'You Only Look Once: Unified, Real-Time Object Detection'. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[38] Thomas B. Sheridan. 'Human–Robot Interaction: Status and Challenges'. In: *Human Factors* 58.4 (2016). PMID: 27098262, pp. 525–532. DOI: 10.1177/0018720816644364. eprint: https://doi.org/10.1177/0018720816644364. URL: https://doi.org/10.1177/0018720816644364.

[39] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2010. ISBN: 9781846286414. URL: https://books.google.no/books?id=jPCAFmE-logC.

[40] M.W. Spong, S. Hutchinson and M. Vidyasagar. *Robot Modeling and Control*. Wiley, 2005. ISBN: 9780471649908. URL: https://books.google.no/books?id=A0OXDwAAQBAJ.

[41] Richard S Sutton, Andrew G Barto et al. *Introduction to reinforcement learning*. MIT press Cambridge, 1998. ISBN: 9780262303842. URL: https://books.google.no/books?id=U57uDwAAQBAJ.

[42] Turja Tuuli and Parviainen Jaana. 'The Use of Affective Care Robots Calls Forth Value-based Consideration'. In: *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*. 2020, pp. 950–955. DOI: 10.1109/RO-MAN47096.2020.9223336.

[43] Lina Van Aerschot and Jaana Parviainen. 'Robots responding to care needs? A multitasking care robot pursued for 25 years, available products offer simple entertainment and instrumental assistance'. In: *Ethics and Information Technology* 22.3 (Sept. 2020), pp. 247–256. ISSN: 1572-8439. DOI: 10.1007/s10676-020-09536-0. URL: https://doi.org/10.1007/s10676-020-09536-0.

[44] Christopher J. C. H. Watkins and Peter Dayan. 'Q-learning'. In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698.